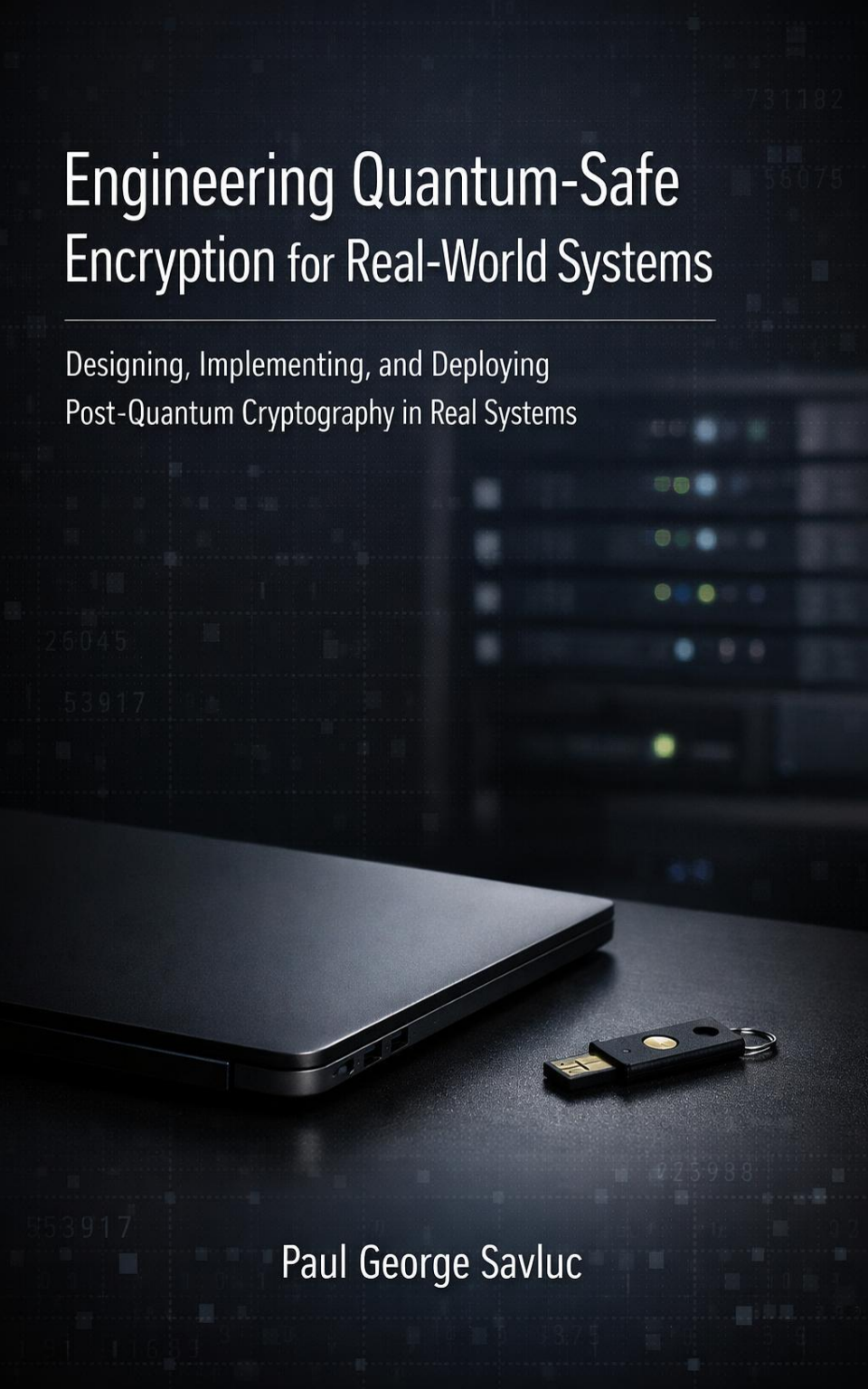
Engineering Quantum-Safe Encryption for Real-World Systems

Designing, Implementing, and Deploying Post-Quantum Cryptography in Real Systems



Paul George Savluc

PAUL SAVLUC

Engineering Quantum-Safe Encryption for Real-World Systems

Designing, Implementing, and Deploying Post-Quantum Cryptography in Real Systems

Copyright © 2025 by Paul Savluc

All rights reserved. No part of this publication may be reproduced, stored or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise without written permission from the publisher. It is illegal to copy this book, post it to a website, or distribute it by any other means without permission.

First edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

To the engineers who build quietly, the defenders who stay invisible, and the students who refuse to accept that security is someone else's job.

To my family and friends for patience during the long nights.

To every team that chose correctness over convenience.

Kind regards,

Paul George Savluc

"Trust is a performance claim. Security is an engineering discipline."

Paul George Savluc

Contents

Foreword vi
Preface viii
Acknowledgments ix
1 Introduction to Post-Quantum Cryptography 1
2 Mathematical Foundations of Lattice-Based Cryptography 4
The mathematics Kyber uses… 4
The ring Kyber lives in 4
The hardness core: noisy linear algebra 5
How Kyber samples small coefficients 6
Encoding message bits into the ring 7
The speed trick: NTT 8
3 A Complete Kyber Pipeline in Python 9
4 Translating the Python Pipeline to RISC-V CPU Code 23
RISC-V project layout 23
RISC-V C code (complete pipeline) 24
kyber_params.h 24
kyber_api.h 25
poly.h 26
fips202.h and fips202.c (SHA3/SHAKE) 28
cbd.c (noise sampling and uniform sampling) 32
ntt.c (NTT, InvNTT, basemul) 36
poly.c (vector ops + encoding/compression) 39

kem.c (Kyber pipeline: keygen, encaps, decaps) 45
test.c (smoke test) 52
Where the "RISC-V translation" becomes real performance work 53
First speed lever: RISC-V "M" extension (mul/div) 53
Second speed lever: RVV (RISC-V Vector) 54
What you now have (and how to use it) 55
5 CRYSTALS-Kyber Algorithm and Implementation Details 57
Key Generation 58
Step 1: Create the public matrix using a seed 58
Step 2: Sample the secret and error values 58
Step 3: Compute the public key polynomial vector 59
Step 4: Output keys and compress 59
Performance note: why it is fast 60
Encapsulation 60
Step 1: Create a fresh message and derive randomness 60
Step 2: Regenerate the matrix A 61
Step 3: Sample the ephemeral secret and errors 61
Step 4: Build the ciphertext components u and v 61
Step 5: Compress ciphertext 62
Performance note 62
Decapsulation 62
Step 1: Recover the message by canceling structure 63

Step 2: Decode by rounding 63
Step 3: Fujisaki-Okamoto verification 63
Failure probability 64
Practical Implementation Notes 64
Symmetric cryptography dominates more than people expect 64
Memory strategies 65
Side-channel resistance is mandatory 65
6 Security Analysis of CRYSTALS-Kyber 66
Underlying Hardness Assumption and Quantum Resistance 66
Known Cryptanalysis and Security Margins 68
Security Against Quantum Adversaries 73
Implementation Security and Considerations 75
7 Comparative Analysis of CRYSTALS-Kyber and Other... 77
CRYSTALS-Dilithium (Post-Quantum Digital Signatures) 77
NTRU (Another Lattice-Based Encryption Scheme) 80
BIKE (Bit-Flipping Key Encapsulation) – Code-Based KEM 82
Other Noteworthy Post-Quantum Schemes (Briefly) 85
8 Adoption and Standardization Status 88
NIST Standardization and Recommendations 88
International and Industry Adoption 91
European Union Initiatives 92
Developments in China, Russia, and Others 94
Overall Readiness and Global Regulations 97

9 Performance Benchmarks of Kyber Across Platforms 99
Desktop CPU performance (Intel and AMD, x86-64) 99
Why vector instructions matter (SIMD and AVX2) 100
Bandwidth and object sizes (public keys, secret keys, ciphertexts) 101
Throughput vs latency (what servers care about) 101
Embedded and mobile performance (Cortex-M class devices) 102
Memory footprint (RAM and flash) 102
Energy cost (mobile reality) 103
Hardware acceleration (FPGA and ASIC friendliness) 103
Comparison with other post-quantum KEMs (engineering view) 104
Bottom line 104
10 Practical Deployment Scenarios and Migration Strategies 105
Deployment Scenarios for CRYSTALS-Kyber 105
Interoperability Challenges 109
Migration Strategies and Best Practices 112
11 Kyber Cheat Sheet 116
What Kyber is 116
CPU and GPU implementation cheat sheet in plain English (one page) 118
CPU (normal computer processor): how to make Kyber fast 118

GPU (graphics processor): when it helps and how to use it 119
12 Conclusion 121
Epilogue 123
Afterword 124

Foreword

Modern encryption is no longer an academic luxury. It is infrastructure. If you ship software, deploy devices, operate networks, or store data, you are already in the cryptography business whether you planned to be or not.

What makes this era unusual is not simply that threats are growing. It is that the assumptions behind widely deployed public-key cryptography are being stress-tested from multiple directions at once: large-scale automation, industrialized exploitation, and the long-horizon risk that quantum computing introduces to today's public-key systems. Post-quantum cryptography is not a sci-fi plot twist. It is a disciplined migration problem, with real constraints: performance, memory, side-channel resistance, supply-chain realities, and the need to keep systems usable.

This book is written for builders. It focuses on how encryption actually behaves inside real systems: on CPUs, GPUs, embedded devices, and in production pipelines where correctness must coexist with deadlines. The goal is not to impress you with symbols. The goal is to help you ship secure systems and understand what you shipped.

If you finish this book, you should be able to read a post-quantum primitive like Kyber, understand its moving parts, reason about performance bottlenecks, and recognize the implementation choices that separate strong security from

accidental vulnerability.

Preface

I wrote this to reduce the distance between "cryptography as theory" and "cryptography as deployed engineering." That distance is where most failures live.

In the field, teams do not lose because the math is weak. They lose because the implementation leaks. They lose because a protocol is integrated incorrectly. They lose because randomness is mishandled, comparisons are not constant-time, keys are logged, builds are not reproducible, or threat models are assumed instead of written down.

Post-quantum cryptography raises the standard. It demands better engineering hygiene because the primitives are more structured, and the performance profile is different. You do not get to treat it like a black box. You have to respect data layout, timing behavior, memory movement, parameterization, and failure modes.

This book uses plain language where possible and numbers where necessary. The intent is clarity over ceremony. If something matters operationally, it is explained in operational terms. If a claim is made, it is tied to an engineering reason.

The reader I had in mind is anyone who must build systems that deserve trust: software engineers, security engineers, embedded developers, infrastructure teams, researchers translating prototypes into products, and leaders who need to make correct decisions under uncertainty.

Acknowledgments

No serious work is solitary, even when it feels like it at 2:00 AM.

I want to acknowledge the broader cryptography and security engineering community that has spent decades turning ideas into standards, proofs into implementations, and research into tools that real people can rely on. Post-quantum cryptography, and Kyber in particular, is the result of open scrutiny, hard criticism, and an ecosystem that understands that trust is earned through adversarial review.

I also acknowledge the engineers who maintain the unglamorous layers: compilers, build systems, testing harnesses, CI pipelines, fuzzers, constant-time libraries, and secure OS primitives. These layers rarely get applause, but they determine whether cryptography stays cryptography when it meets reality.

Finally, thank you to the teams and collaborators who continually sharpen my thinking through challenges, design reviews, and the simple discipline of asking, "How would this fail?"

1

Introduction to Post-Quantum Cryptography

The advent of quantum computing poses a serious threat to classical cryptosystems like RSA and ECC, which rely on the difficulty of factoring and discrete logarithms. Shor's algorithm, running on a sufficiently powerful quantum computer, can efficiently break RSA/ECC, endangering the security of virtually all current encrypted communications. This looming threat has spurred the development of **post-quantum cryptography (PQC)** – new cryptographic algorithms designed to resist quantum attacks. In 2016, the U.S. National Institute of Standards and Technology (NIST) launched an open competition to identify quantum-resistant algorithms, focusing on public-key encryption (key establishment) and digital signatures. After multiple evaluation rounds, NIST announced its first set of PQC standards in 2022, selecting four algorithms (one KEM and three digital signatures) for standardization. Notably, three of the four rely on structured lattices, reflecting NIST's view that lattice-based cryptography is the most promising approach

for quantum safety.

One of these new standards is **CRYSTALS-Kyber**, an asymmetric encryption algorithm for establishing shared secrets over public networks. Kyber is a **Key Encapsulation Mechanism (KEM)**: rather than directly encrypting arbitrary messages, it encapsulates a randomly chosen symmetric key that the communicating parties can then use for conventional encryption (e.g., AES). In a KEM exchange, one party uses the recipient's public key to produce a ciphertext that "encapsulates" a random secret; the recipient uses their private key to **decapsulate** the ciphertext and recover that secret. This differs from a Diffie–Hellman key exchange where both parties jointly compute a shared secret. **Figure 1.1** illustrates this contrast: in a KEM, the sender chooses the secret and encrypts it to the receiver's public key, whereas Diffie–Hellman has both sides interact to derive a secret without prior encryption.

Figure 1.1: A conceptual illustration comparing a Key Encapsulation Mechanism (KEM) approach (left) with a Diffie–Hellman (DH) key exchange (right). In a KEM, the sender uses the recipient's public key to encapsulate a random secret into a ciphertext, which the recipient decapsulates with their private key. In DH, both parties contribute to the secret through a mathematical exchange.

Kyber's importance stems from its strong security and efficiency. It was the sole KEM selected by NIST for primary standardization, beating out other finalists like NTRU and SABER due to its blend of small key sizes, high speed, and trusted hardness assumptions. CRYSTALS-Kyber is part of the "Cryptographic Suite for Algebraic Lattices" (CRYSTALS), which also includes the Dilithium digital signature algorithm. The name "CRYSTALS" highlights that these schemes are built on **algebraic lattices**, a mathematical structure believed to be

resistant to quantum attacks. Indeed, lattice-based cryptography is currently seen as the best option for securing data against future quantum adversaries. In the following chapters, we will delve deeply into the mathematics of lattice cryptography and Kyber, detail Kyber's design and implementation, compare it with other post-quantum algorithms, analyze its security, and discuss the ongoing global efforts to adopt these new cryptographic standards.

2

Mathematical Foundations of Lattice-Based Cryptography

The mathematics Kyber uses…

No mysticism. Just structured arithmetic.

The ring Kyber lives in

Kyber works with polynomials that have 256 coefficients. The highest power is 255.

Each coefficient is an integer taken modulo 3329. That means every coefficient is always reduced into the range 0 through 3328.

So a Kyber polynomial is basically a length 256 list of integers, each one living in 0 to 3328.

Addition works like this:

- Add the two coefficients in the same position.
- Reduce the result modulo 3329.

- Do that for all 256 positions.

Multiplication works like polynomial convolution, but with a special wraparound rule.

The wraparound rule is:

- Any time a multiplication produces a term with power 256 or higher, you fold it back into the low powers.
- When you fold it back, you flip the sign.

Concrete folding intuition:

- A term at power 256 becomes the same as power 0 but negated.
- A term at power 257 becomes the same as power 1 but negated.
- In general, power 256 plus i becomes power i but negated.

This is called negacyclic reduction. It is the reason Kyber's ring acts like a circular buffer of length 256, except the wrap adds a sign flip.

The hardness core: noisy linear algebra

Kyber is built on a problem called Module LWE. In plain words, it is "solve a linear system that has noise added, where the unknowns are small".

Kyber uses:

- A public matrix of polynomials, with size k by k.
- A secret vector of polynomials, length k, with small coeffi-

cients.

- A noise vector of polynomials, length k, also with small coefficients.

The public output is computed by:

- Multiply the public matrix by the secret vector.
- Add the noise vector.
- Reduce everything in the ring so coefficients stay modulo 3329 and powers wrap using the negacyclic rule.

Why noise matters:

- If the noise were exactly zero, the public output would behave like a clean linear system and attackers could solve for the secret.
- With noise, the attacker sees a linear system that is slightly wrong in many places.
- Recovering the secret becomes a lattice decoding problem, which is believed to be hard at Kyber's parameters.

How Kyber samples small coefficients

Kyber needs secrets and noise that are small integers, not uniform random numbers from 0 to 3328.

A common method is a centered binomial distribution.

Here is the idea in words:

- Pick eta random bits.
- Count how many are 1.
- Pick another eta random bits.

- Count how many are 1.
- Subtract the second count from the first count.

That gives a small integer in the range negative eta through positive eta.

Then Kyber stores that result as a coefficient modulo 3329. So negative values get represented as large numbers near 3329, because modulo arithmetic wraps them around.

This "small but random" noise is the fuel that makes the underlying lattice problem hard.

Encoding message bits into the ring

Kyber embeds a 256 bit message into one polynomial with 256 coefficients.

Each bit is mapped into one coefficient using two values that are very far apart modulo 3329:

- Bit 0 becomes 0.
- Bit 1 becomes the integer floor of 3329 divided by 2, which is 1664.

So you can think of the message polynomial as:

- Coefficients are either 0 or 1664.

Later, during decryption, the receiver computes a value that should be close to 0 or close to 1664, plus a small amount of noise. Then the receiver rounds it back to the nearest of those two targets to recover the bit.

The key point is separation:

- 0 and 1664 are far apart.
- The noise is small.
- So rounding is reliable with extremely high probability.

The speed trick: NTT

Naive polynomial multiplication for length 256 is expensive because it needs on the order of 256 times 256 coefficient operations.

Kyber speeds this up using the Number Theoretic Transform, or NTT. It is like an FFT, but everything happens modulo 3329.

High level flow:

1. Transform polynomial a into the NTT domain.
2. Transform polynomial b into the NTT domain.
3. Multiply point by point in the NTT domain.
4. Inverse transform back to get the product polynomial.

For size 256:

- The transform has 8 stages because 256 is 2 to the power 8.
- Each stage is built from butterfly operations, which are small arithmetic kernels that mix pairs of values.
- A typical transform at this size performs about 1024 butterflies.

This is why optimized Kyber implementations obsess over NTT kernels. The NTT dominates runtime, so faster butterflies mean faster key exchange on CPUs, microcontrollers, and GPUs.

3

A Complete Kyber Pipeline in Python

This Python application is written as a "professional reference pipeline": clean stages, correct integer compression, and the full KEM structure (keygen, encaps, decaps). It uses Python's hashlib SHAKE/SHA3.

Save as kyber_ref.py:

```
#!/usr/bin/env python3
from __future__ import annotations

from dataclasses import dataclass
from typing import List, Tuple, Dict
import hashlib
import os


# ===== Constants =====
q = 3329
n = 256
zeta = 17
inv2 = (q + 1) // 2
```

```
ETA2 = 2


# ===== Bit reversal for 7-bit values (0..127) =====
def brv7(x: int) -> int:
    b = f"{x:07b}"
    return int(b[::-1], 2)

BRV = [brv7(i) for i in range(128)]


# ===== Precompute twiddle helpers consistent across
NTT/InvNTT/BaseMul =====
ZETA_POW = [0] * 128
for i in range(1, 128):
    ZETA_POW[i] = pow(zeta, BRV[i], q)

ZETA_ODD = [0] * 128
for i in range(128):
    ZETA_ODD[i] = pow(zeta, 2 * BRV[i] + 1, q)


# ===== Params =====
@dataclass(frozen=True)
class KyberParams:
    name: str
    k: int
    du: int
    dv: int
    eta1: int

KYBER512  = KyberParams("Kyber512",  k=2, du=10,
dv=4, eta1=3)
KYBER768  = KyberParams("Kyber768",  k=3, du=10,
dv=4, eta1=2)
KYBER1024 = KyberParams("Kyber1024", k=4, du=11,
dv=5, eta1=2)
```

```
# ===== Compression =====
def compress_coeff(x: int, d: int) -> int:
    return (((x << d) + (q // 2)) // q) & ((1 << d) -
    1)

def decompress_coeff(y: int, d: int) -> int:
    return (q * y + (1 << (d - 1))) >> d


# ===== Bit packing (LSB-first) =====
def pack_bits(values: List[int], width: int) -> bytes:
    out = bytearray()
    acc = 0
    acc_bits = 0
    mask = (1 << width) - 1
    for v in values:
        v &= mask
        acc |= (v << acc_bits)
        acc_bits += width
        while acc_bits >= 8:
            out.append(acc & 0xFF)
            acc >>= 8
            acc_bits -= 8
    if acc_bits:
        out.append(acc & 0xFF)
    return bytes(out)

def unpack_bits(data: bytes, width: int, count: int)
-> List[int]:
    vals = []
    acc = 0
    acc_bits = 0
    mask = (1 << width) - 1
    idx = 0
    for _ in range(count):
```

```
        while acc_bits < width:
            acc |= data[idx] << acc_bits
            acc_bits += 8
            idx += 1
        vals.append(acc & mask)
        acc >>= width
        acc_bits -= width
    return vals


# ===== SHAKE streaming helper =====
class ShakeStream:
    def __init__(self, shake_obj):
        self._shake = shake_obj
        self._buf = b""
        self._pos = 0
        self._want = 0

    def read(self, nbytes: int) -> bytes:
        need = self._pos + nbytes
        if need > len(self._buf):
            self._want = max(need, max(1024, 2 *
            len(self._buf)))
            self._buf = self._shake.digest(self._want)
        out = self._buf[self._pos:self._pos + nbytes]
        self._pos += nbytes
        return out


# ===== XOF / PRF =====
def XOF_stream(rho: bytes, j: int, i: int) ->
ShakeStream:
    h = hashlib.shake_128()
    h.update(rho + bytes([j & 0xFF, i & 0xFF]))
    return ShakeStream(h)

def PRF_stream(sigma: bytes, nonce: int) ->
```

```
ShakeStream:
    h = hashlib.shake_256()
    h.update(sigma + bytes([nonce & 0xFF]))
    return ShakeStream(h)


# ===== Sampling =====
def sample_uniform(stream: ShakeStream) -> List[int]:
    coeffs = []
    while len(coeffs) < n:
        b0, b1, b2 = stream.read(3)
        d1 = b0 + 256 * (b1 & 0x0F)
        d2 = (b1 >> 4) + 16 * b2
        if d1 < q:
            coeffs.append(d1)
        if len(coeffs) < n and d2 < q:
            coeffs.append(d2)
    return coeffs

def cbd_from_bytes(a: bytes, eta: int) -> List[int]:
    if len(a) != 64 * eta:
        raise ValueError("CBD input must be 64*eta
        bytes.")
    bits = []
    for byte in a:
        for j in range(8):
            bits.append((byte >> j) & 1)

    coeffs = []
    idx = 0
    for _ in range(n):
        s1 = sum(bits[idx:idx + eta])
        s2 = sum(bits[idx + eta:idx + 2 * eta])
        idx += 2 * eta
        coeffs.append((s1 - s2) % q)
    return coeffs
```

```
# ===== Poly / Vec / Matrix =====
@dataclass(frozen=True)
class Poly:
    cs: Tuple[int, ...]

    @staticmethod
    def zero() -> "Poly":
        return Poly((0,) * n)

    def add(self, other: "Poly") -> "Poly":
        return Poly(tuple((a + b) % q for a, b in
        zip(self.cs, other.cs)))

    def sub(self, other: "Poly") -> "Poly":
        return Poly(tuple((a - b) % q for a, b in
        zip(self.cs, other.cs)))

    def compress(self, d: int) -> "Poly":
        return Poly(tuple(compress_coeff(c, d) for c
        in self.cs))

    def decompress(self, d: int) -> "Poly":
        return Poly(tuple(decompress_coeff(c, d) for
        c in self.cs))

    def encode(self, width: int) -> bytes:
        return pack_bits(list(self.cs), width)

    @staticmethod
    def decode(data: bytes, width: int) -> "Poly":
        return Poly(tuple(unpack_bits(data, width,
        n)))

    def ntt(self) -> "Poly":
        a = list(self.cs)
        layer = n // 2
```

```
        zi = 0
        while layer >= 2:
            step = 2 * layer
            for offset in range(0, n - layer, step):
                zi += 1
                z = ZETA_POW[zi]
                for j in range(offset, offset +
                layer):
                    t = (z * a[j + layer]) % q
                    a[j + layer] = (a[j] - t) % q
                    a[j] = (a[j] + t) % q
            layer //= 2
        return Poly(tuple(a))

    def inv_ntt(self) -> "Poly":
        a = list(self.cs)
        layer = 2
        zi = n // 2
        while layer < n:
            step = 2 * layer
            for offset in range(0, n - layer, step):
                zi -= 1
                z = ZETA_POW[zi]
                for j in range(offset, offset +
                layer):
                    t = (a[j + layer] - a[j]) % q
                    a0 = (a[j] + a[j + layer]) % q
                    a[j] = (inv2 * a0) % q
                    a[j + layer] = (inv2 * z * t) % q
            layer *= 2
        return Poly(tuple(a))

    def basemul_ntt(self, other: "Poly") -> "Poly":
        a = self.cs
        b = other.cs
        out = [0] * n
        for i in range(0, n, 2):
```

```
            a1, a2 = a[i], a[i + 1]
            b1, b2 = b[i], b[i + 1]
            z = ZETA_ODD[i // 2]
            out[i] = (a1 * b1 + z * a2 * b2) % q
            out[i + 1] = (a2 * b1 + a1 * b2) % q
        return Poly(tuple(out))


@dataclass(frozen=True)
class Vec:
    ps: Tuple[Poly, ...]

    def ntt(self) -> "Vec":
        return Vec(tuple(p.ntt() for p in self.ps))

    def inv_ntt(self) -> "Vec":
        return Vec(tuple(p.inv_ntt() for p in
        self.ps))

    def add(self, other: "Vec") -> "Vec":
        return Vec(tuple(a.add(b) for a, b in
        zip(self.ps, other.ps)))

    def compress(self, d: int) -> "Vec":
        return Vec(tuple(p.compress(d) for p in
        self.ps))

    def decompress(self, d: int) -> "Vec":
        return Vec(tuple(p.decompress(d) for p in
        self.ps))

    def encode(self, width: int) -> bytes:
        coeffs = []
        for p in self.ps:
            coeffs.extend(p.cs)
        return pack_bits(coeffs, width)
```

```
    @staticmethod
    def decode(data: bytes, k: int, width: int) ->
    "Vec":
        vals = unpack_bits(data, width, k * n)
        polys = []
        for i in range(k):
            polys.append(Poly(tuple(vals[i * n:(i +
            1) * n])))
        return Vec(tuple(polys))

    def dot_ntt(self, other: "Vec") -> Poly:
        acc = Poly.zero()
        for a, b in zip(self.ps, other.ps):
            acc = acc.add(a.basemul_ntt(b))
        return acc


@dataclass(frozen=True)
class Matrix:
    rows: Tuple[Tuple[Poly, ...], ...]

    def transpose(self) -> "Matrix":
        k = len(self.rows)
        cols = []
        for j in range(k):
            cols.append(tuple(self.rows[i][j] for i
            in range(k)))
        return Matrix(tuple(cols))

    def mul_ntt(self, v: Vec) -> Vec:
        out = []
        for row in self.rows:
            out.append(Vec(tuple(row)).dot_ntt(v))
        return Vec(tuple(out))


# ===== Hashing primitives =====
```

```
def H(m: bytes) -> bytes:
    return hashlib.sha3_256(m).digest()

def G(m: bytes) -> Tuple[bytes, bytes]:
    h = hashlib.sha3_512(m).digest()
    return h[:32], h[32:]

def KDF(m: bytes) -> bytes:
    return hashlib.shake_256(m).digest(32)


# ===== Expand A, sample noise =====
def sample_matrix(rho: bytes, k: int) -> Matrix:
    rows = []
    for i in range(k):
        row = []
        for j in range(k):
            row.append(Poly(tuple(sample_uniform(XOF_stream(rho,
            j, i)))))
        rows.append(tuple(row))
    return Matrix(tuple(rows))

def sample_noise_vec(seed: bytes, eta: int, offset:
int, k: int) -> Vec:
    polys = []
    for i in range(k):
        raw = PRF_stream(seed, i + offset).read(64 *
        eta)
        polys.append(Poly(tuple(cbd_from_bytes(raw,
        eta))))
    return Vec(tuple(polys))


# ===== CPA PKE =====
def inner_keygen(seed32: bytes, params: KyberParams)
-> Tuple[bytes, bytes]:
    rho, sigma = G(seed32)
```

```
    A = sample_matrix(rho, params.k)

    s = sample_noise_vec(sigma, params.eta1, 0,
    params.k)
    e = sample_noise_vec(sigma, params.eta1,
    params.k, params.k)

    s_hat = s.ntt()
    e_hat = e.ntt()
    t_hat = A.mul_ntt(s_hat).add(e_hat)

    pk = t_hat.encode(12) + rho
    sk = s_hat.encode(12)
    return pk, sk

def inner_enc(pk: bytes, msg32: bytes, coins32:
bytes, params: KyberParams) -> bytes:
    t_hat = Vec.decode(pk[:-32], params.k, 12)
    rho = pk[-32:]
    A = sample_matrix(rho, params.k)

    r  = sample_noise_vec(coins32, params.eta1, 0,
    params.k)
    e1 = sample_noise_vec(coins32, ETA2, params.k,
    params.k)
    e2 = sample_noise_vec(coins32, ETA2, 2 *
    params.k, 1).ps[0]

    r_hat = r.ntt()
    u = A.transpose().mul_ntt(r_hat).inv_ntt().add(e1)

    m_bits = unpack_bits(msg32, 1, n)
    m_poly = Poly(tuple(m_bits)).decompress(1)

    v =
    t_hat.dot_ntt(r_hat).inv_ntt().add(e2).add(m_poly)
```

```
    c1 = u.compress(params.du).encode(params.du)
    c2 = v.compress(params.dv).encode(params.dv)
    return c1 + c2

def inner_dec(sk: bytes, ct: bytes, params:
KyberParams) -> bytes:
    split = (params.du * params.k * n) // 8
    c1, c2 = ct[:split], ct[split:]

    u = Vec.decode(c1, params.k,
    params.du).decompress(params.du)
    v = Poly.decode(c2,
    params.dv).decompress(params.dv)
    s_hat = Vec.decode(sk, params.k, 12)

    m =
    v.sub(s_hat.dot_ntt(u.ntt()).inv_ntt()).compress(1)
    return pack_bits(list(m.cs), 1)


# ===== KEM wrapper (FO transform) =====
def keygen(seed64: bytes, params: KyberParams) ->
Tuple[bytes, bytes]:
    pk, sk_cpa = inner_keygen(seed64[:32], params)
    hpk = H(pk)
    z = seed64[32:]
    sk = sk_cpa + pk + hpk + z
    return pk, sk

def encaps(pk: bytes, seed32: bytes, params:
KyberParams) -> Tuple[bytes, bytes]:
    m = H(seed32)
    Kbar, coins = G(m + H(pk))
    ct = inner_enc(pk, m, coins, params)
    K = KDF(Kbar + H(ct))
    return ct, K
```

```
def decaps(sk: bytes, ct: bytes, params: KyberParams)
-> bytes:
    sk_cpa_len = (12 * params.k * n) // 8
    pk_len = (12 * params.k * n) // 8 + 32

    sk_cpa = sk[:sk_cpa_len]
    pk = sk[sk_cpa_len:sk_cpa_len + pk_len]
    hpk = sk[sk_cpa_len + pk_len: sk_cpa_len + pk_len
    + 32]
    z = sk[sk_cpa_len + pk_len + 32: sk_cpa_len +
    pk_len + 64]

    m2 = inner_dec(sk_cpa, ct, params)
    Kbar2, coins2 = G(m2 + hpk)
    ct2 = inner_enc(pk, m2, coins2, params)

    K1 = KDF(Kbar2 + H(ct))
    K2 = KDF(z + H(ct))
    # Python compare_digest is constant-time-ish for
    bytes
    ok = hashlib.compare_digest(ct, ct2)
    return K1 if ok else K2


def self_test():
    for params in (KYBER512, KYBER768, KYBER1024):
        pk, sk = keygen(os.urandom(64), params)
        ct, ss1 = encaps(pk, os.urandom(32), params)
        ss2 = decaps(sk, ct, params)
        assert ss1 == ss2, params.name
    print("OK: Kyber512/Kyber768/Kyber1024")

if __name__ == "__main__":
    self_test()
```

That is a complete pipeline: expand matrix AAA, sample noise,

NTT, dot-products in NTT domain, invNTT, compress/pack, then the KEM re-encrypt-and-compare step.

4

Translating the Python Pipeline to RISC-V CPU Code

"RISC-V CPU code" should mean: code that compiles cleanly for rv64gc (and optionally rv64gcv for vector). The standard way to deliver that is:

- Portable scalar C that is correct and constant-time friendly in structure.
- Optional RVV kernels for NTT and basemul, guarded by #ifdef __riscv_vector.

Below is a complete, minimal, professional layout. It is still "reference-grade" in the sense that it prioritizes correctness and readability. It is designed so you can replace the scalar NTT and basemul with an RVV-optimized version later without changing the pipeline.

RISC-V project layout

```
kyber_riscv/
  kyber_params.h
  kyber_api.h
  poly.h
  poly.c
  ntt.c
  cbd.c
  fips202.h
  fips202.c
  kem.c
  test.c
```

RISC-V C code (complete pipeline)

kyber_params.h

```
#pragma once
#include <stdint.h>

#define KYBER_Q 3329
#define KYBER_N 256
#define KYBER_ZETA 17

// Choose one parameter set by defining KYBER_K,
KYBER_ETA1, KYBER_DU, KYBER_DV
// Example Kyber768:
#ifndef KYBER_K
#define KYBER_K 3
#endif

#ifndef KYBER_ETA1
```

```
#define KYBER_ETA1 2
#endif

#ifndef KYBER_ETA2
#define KYBER_ETA2 2
#endif

#ifndef KYBER_DU
#define KYBER_DU 10
#endif

#ifndef KYBER_DV
#define KYBER_DV 4
#endif

// Sizes
#define POLY_BYTES_12 (12 * KYBER_N / 8)          //
384
#define VEC_BYTES_12  (KYBER_K * POLY_BYTES_12)   //
k*384
#define PK_BYTES      (VEC_BYTES_12 + 32)         //
tHat + rho
#define SKCPA_BYTES   (VEC_BYTES_12)              //
sHat
#define C1_BYTES      (KYBER_DU * KYBER_K * KYBER_N /
8)
#define C2_BYTES      (KYBER_DV * KYBER_N / 8)
#define CT_BYTES      (C1_BYTES + C2_BYTES)
#define SS_BYTES      32
```

kyber_api.h

```
#pragma once
#include <stdint.h>
#include "kyber_params.h"

int kyber_keygen(uint8_t pk[PK_BYTES], uint8_t
sk[SKCPA_BYTES + PK_BYTES + 64], const uint8_t
seed64[64]);
int kyber_encaps(uint8_t ct[CT_BYTES], uint8_t
ss[SS_BYTES], const uint8_t pk[PK_BYTES], const
uint8_t seed32[32]);
int kyber_decaps(uint8_t ss[SS_BYTES], const uint8_t
ct[CT_BYTES], const uint8_t sk[SKCPA_BYTES + PK_BYTES
+ 64]);
```

poly.h

```
#pragma once
#include <stdint.h>
#include "kyber_params.h"

typedef struct {
    int16_t c[KYBER_N]; // coefficients in [0, q)
} poly;

typedef struct {
    poly v[KYBER_K];
} polyvec;

typedef struct {
    poly m[KYBER_K][KYBER_K];
} polymat;
```

```
// NTT domain operations
void poly_ntt(poly* a);
void poly_invntt(poly* a);
void poly_basemul_ntt(poly* r, const poly* a, const
poly* b);

// Vector helpers
void polyvec_ntt(polyvec* v);
void polyvec_invntt(polyvec* v);
void polyvec_add(polyvec* r, const polyvec* a, const
polyvec* b);
void polyvec_dot_ntt(poly* r, const polyvec* a, const
polyvec* b);

// Matrix-vector multiply in NTT domain: r = A * v
void polymat_mul_ntt(polyvec* r, const polymat* A,
const polyvec* v);
void polymat_transpose(polymat* At, const polymat* A);

// Compression/encoding
void poly_compress_bits(uint8_t* out, const poly* a,
int d);
void poly_decompress_bits(poly* a, const uint8_t* in,
int d);

void polyvec_compress_bits(uint8_t* out, const
polyvec* v, int d);
void polyvec_decompress_bits(polyvec* v, const
uint8_t* in, int d);

// Encode/decode width=12 for NTT-domain secrets and
public key parts
void poly_encode_12(uint8_t out[POLY_BYTES_12], const
poly* a);
void poly_decode_12(poly* a, const uint8_t
in[POLY_BYTES_12]);
```

```
void polyvec_encode_12(uint8_t out[VEC_BYTES_12],
const polyvec* v);
void polyvec_decode_12(polyvec* v, const uint8_t
in[VEC_BYTES_12]);
```

fips202.h and fips202.c (SHA3/SHAKE)

This is a compact Keccak-f[1600] SHAKE implementation sufficient for Kyber-style pipelines.

fips202.h:

```
#pragma once
#include <stdint.h>
#include <stddef.h>

void sha3_256(uint8_t out[32], const uint8_t* in,
size_t inlen);
void sha3_512(uint8_t out[64], const uint8_t* in,
size_t inlen);

void shake128(uint8_t* out, size_t outlen, const
uint8_t* in, size_t inlen);
void shake256(uint8_t* out, size_t outlen, const
uint8_t* in, size_t inlen);
```

fips202.c (compact, readable, not micro-optimized):

```
#include "fips202.h"
#include <string.h>
#include <stdint.h>
```

```
static inline uint64_t rol64(uint64_t x, unsigned r) {
    return (x << r) | (x >> (64 - r));
}

static const uint64_t RC[24] = {
    0x0000000000000001ULL, 0x0000000000008082ULL,
    0x800000000000808aULL, 0x8000000080008000ULL,
    0x000000000000808bULL, 0x0000000080000001ULL,
    0x8000000080008081ULL, 0x8000000000008009ULL,
    0x000000000000008aULL, 0x0000000000000088ULL,
    0x0000000080008009ULL, 0x000000008000000aULL,
    0x000000008000808bULL, 0x800000000000008bULL,
    0x8000000000008089ULL, 0x8000000000008003ULL,
    0x8000000000008002ULL, 0x8000000000000080ULL,
    0x000000000000800aULL, 0x800000008000000aULL,
    0x8000000080008081ULL, 0x8000000000008080ULL,
    0x0000000080000001ULL, 0x8000000080008008ULL
};

static const int R[5][5] = {
    { 0, 36,  3, 41, 18},
    { 1, 44, 10, 45,  2},
    {62,  6, 43, 15, 61},
    {28, 55, 25, 21, 56},
    {27, 20, 39,  8, 14}
};

static void keccak_f1600(uint64_t s[25]) {
    for (int round = 0; round < 24; round++) {
        uint64_t C[5], D[5], B[25];

        for (int x = 0; x < 5; x++) {
            C[x] = s[x] ^ s[x + 5] ^ s[x + 10] ^ s[x
            + 15] ^ s[x + 20];
        }
        for (int x = 0; x < 5; x++) {
```

```
            D[x] = C[(x + 4) % 5] ^ rol64(C[(x + 1) %
            5], 1);
        }
        for (int i = 0; i < 25; i++) s[i] ^= D[i % 5];

        for (int y = 0; y < 5; y++) {
            for (int x = 0; x < 5; x++) {
                B[y * 5 + x] = rol64(s[y * 5 + x],
                R[x][y]);
            }
        }

        for (int y = 0; y < 5; y++) {
            for (int x = 0; x < 5; x++) {
                s[y * 5 + x] = B[y * 5 + x] ^ ((~B[y
                * 5 + ((x + 1) % 5)]) & B[y * 5 + ((x
                + 2) % 5)]);
            }
        }

        s[0] ^= RC[round];
    }
}

static void keccak_absorb(uint64_t s[25], unsigned
rate, const uint8_t* in, size_t inlen, uint8_t pad) {
    memset(s, 0, 25 * sizeof(uint64_t));

    while (inlen >= rate) {
        for (unsigned i = 0; i < rate / 8; i++) {
            uint64_t t = 0;
            for (int b = 0; b < 8; b++) t |=
            ((uint64_t)in[8*i + b]) << (8*b);
            s[i] ^= t;
        }
        keccak_f1600(s);
        in += rate;
```

```
        inlen -= rate;
    }

    uint8_t tmp[200];
    memset(tmp, 0, rate);
    memcpy(tmp, in, inlen);
    tmp[inlen] = pad;
    tmp[rate - 1] |= 0x80;

    for (unsigned i = 0; i < rate / 8; i++) {
        uint64_t t = 0;
        for (int b = 0; b < 8; b++) t |=
        ((uint64_t)tmp[8*i + b]) << (8*b);
        s[i] ^= t;
    }
}

static void keccak_squeeze(uint8_t* out, size_t
outlen, uint64_t s[25], unsigned rate) {
    while (outlen) {
        keccak_f1600(s);
        size_t block = (outlen < rate) ? outlen :
        rate;
        for (size_t i = 0; i < block; i++) {
            out[i] = (uint8_t)((s[i / 8] >> (8 * (i %
            8))) & 0xFF);
        }
        out += block;
        outlen -= block;
    }
}

void shake128(uint8_t* out, size_t outlen, const
uint8_t* in, size_t inlen) {
    uint64_t s[25];
    keccak_absorb(s, 168, in, inlen, 0x1F);
    keccak_squeeze(out, outlen, s, 168);
```

```
}

void shake256(uint8_t* out, size_t outlen, const
uint8_t* in, size_t inlen) {
    uint64_t s[25];
    keccak_absorb(s, 136, in, inlen, 0x1F);
    keccak_squeeze(out, outlen, s, 136);
}

void sha3_256(uint8_t out[32], const uint8_t* in,
size_t inlen) {
    uint64_t s[25];
    keccak_absorb(s, 136, in, inlen, 0x06);
    keccak_squeeze(out, 32, s, 136);
}

void sha3_512(uint8_t out[64], const uint8_t* in,
size_t inlen) {
    uint64_t s[25];
    keccak_absorb(s, 72, in, inlen, 0x06);
    keccak_squeeze(out, 64, s, 72);
}
```

cbd.c (noise sampling and uniform sampling)

```
#include "kyber_params.h"
#include "fips202.h"
#include "poly.h"
#include <string.h>
#include <stdint.h>

static void xof_stream(uint8_t* out, size_t outlen,
const uint8_t rho[32], uint8_t j, uint8_t i) {
```

```
    uint8_t in[34];
    memcpy(in, rho, 32);
    in[32] = j;
    in[33] = i;
    shake128(out, outlen, in, 34);
}

static void prf_stream(uint8_t* out, size_t outlen,
const uint8_t sigma[32], uint8_t nonce) {
    uint8_t in[33];
    memcpy(in, sigma, 32);
    in[32] = nonce;
    shake256(out, outlen, in, 33);
}

// Rejection sampling for one polynomial from SHAKE
stream
static void sample_uniform_poly(poly* a, const
uint8_t rho[32], uint8_t j, uint8_t i) {
    // Over-generate and then accept < q until filled.
    // For simplicity: produce a few KB and loop.
    Reference implementations stream.
    uint8_t buf[4096];
    xof_stream(buf, sizeof(buf), rho, j, i);

    int filled = 0;
    for (size_t p = 0; p + 2 < sizeof(buf) && filled
    < KYBER_N; p += 3) {
        uint16_t d1 = (uint16_t)buf[p] |
        ((uint16_t)(buf[p+1] & 0x0F) << 8);
        uint16_t d2 = (uint16_t)(buf[p+1] >> 4) |
        ((uint16_t)buf[p+2] << 4);
        if (d1 < KYBER_Q) a->c[filled++] =
        (int16_t)d1;
        if (filled < KYBER_N && d2 < KYBER_Q)
        a->c[filled++] = (int16_t)d2;
    }
```

```
    // If not filled (rare with 4096 bytes),
    regenerate with larger buffer or loop.
    while (filled < KYBER_N) {
        uint8_t extra[1024];
        xof_stream(extra, sizeof(extra), rho, j,
        (uint8_t)(i + 1));
        for (size_t p = 0; p + 2 < sizeof(extra) &&
        filled < KYBER_N; p += 3) {
            uint16_t d1 = (uint16_t)extra[p] |
            ((uint16_t)(extra[p+1] & 0x0F) << 8);
            uint16_t d2 = (uint16_t)(extra[p+1] >> 4)
            | ((uint16_t)extra[p+2] << 4);
            if (d1 < KYBER_Q) a->c[filled++] =
            (int16_t)d1;
            if (filled < KYBER_N && d2 < KYBER_Q)
            a->c[filled++] = (int16_t)d2;
        }
    }
}

// CBD sampler
static void cbd_poly(poly* a, const uint8_t seed[32],
uint8_t nonce, int eta) {
    uint8_t buf[64 * 3]; // max eta=3 in Kyber sets
    here
    prf_stream(buf, (size_t)(64 * eta), seed, nonce);

    // Interpret bits LSB-first
    int bitpos = 0;
    for (int i = 0; i < KYBER_N; i++) {
        int s1 = 0, s2 = 0;
        for (int j = 0; j < eta; j++) {
            int byte = (bitpos >> 3);
            int off  = (bitpos & 7);
            s1 += (buf[byte] >> off) & 1;
            bitpos++;
```

```
        }
        for (int j = 0; j < eta; j++) {
            int byte = (bitpos >> 3);
            int off  = (bitpos & 7);
            s2 += (buf[byte] >> off) & 1;
            bitpos++;
        }
        int x = s1 - s2;
        x %= KYBER_Q;
        if (x < 0) x += KYBER_Q;
        a->c[i] = (int16_t)x;
    }
}

void kyber_expand_matrix(polymat* A, const uint8_t
rho[32]) {
    for (int i = 0; i < KYBER_K; i++) {
        for (int j = 0; j < KYBER_K; j++) {
            sample_uniform_poly(&A->m[i][j], rho,
            (uint8_t)j, (uint8_t)i);
        }
    }
}

void kyber_sample_noise(polyvec* s, polyvec* e, const
uint8_t sigma[32]) {
    uint8_t nonce = 0;
    for (int i = 0; i < KYBER_K; i++)
    cbd_poly(&s->v[i], sigma, nonce++, KYBER_ETA1);
    for (int i = 0; i < KYBER_K; i++)
    cbd_poly(&e->v[i], sigma, nonce++, KYBER_ETA1);
}

void kyber_sample_enc_noise(polyvec* r, polyvec* e1,
poly* e2, const uint8_t coins[32]) {
    uint8_t nonce = 0;
```

```
    for (int i = 0; i < KYBER_K; i++)
    cbd_poly(&r->v[i], coins, nonce++, KYBER_ETA1);
    for (int i = 0; i < KYBER_K; i++)
    cbd_poly(&e1->v[i], coins, nonce++, KYBER_ETA2);
    cbd_poly(e2, coins, nonce++, KYBER_ETA2);
}
```

Note: kyber_expand_matrix and the noise samplers are used by the main pipeline in kem.c.

ntt.c (NTT, InvNTT, basemul)

This implementation matches the Python structure by computing the same twiddle sequence from bit-reversal, so you do not need a hardcoded zetas[] table.

```
#include "poly.h"
#include <stdint.h>
#include <string.h>

static int brv7(int x) {
    int r = 0;
    for (int i = 0; i < 7; i++) {
        r = (r << 1) | ((x >> i) & 1);
    }
    return r;
}

static int16_t modq(int32_t x) {
    x %= KYBER_Q;
    if (x < 0) x += KYBER_Q;
    return (int16_t)x;
}
```

```
static int16_t powmod(int16_t a, int e) {
    int32_t r = 1;
    int32_t x = a;
    while (e > 0) {
        if (e & 1) r = (r * x) % KYBER_Q;
        x = (x * x) % KYBER_Q;
        e >>= 1;
    }
    return (int16_t)r;
}

static int16_t ZETA_POW[128];
static int16_t ZETA_ODD[128];
static int inited = 0;

static void init_zetas(void) {
    if (inited) return;
    ZETA_POW[0] = 0;
    for (int i = 1; i < 128; i++) {
        int e = brv7(i);
        ZETA_POW[i] = powmod(KYBER_ZETA, e);
    }
    for (int i = 0; i < 128; i++) {
        int e = 2 * brv7(i) + 1;
        ZETA_ODD[i] = powmod(KYBER_ZETA, e);
    }
    inited = 1;
}

void poly_ntt(poly* a) {
    init_zetas();
    int16_t* x = a->c;

    int layer = KYBER_N / 2; // 128
    int zi = 0;
    while (layer >= 2) {
        int step = 2 * layer;
```

```
        for (int offset = 0; offset < KYBER_N -
        layer; offset += step) {
            zi++;
            int16_t z = ZETA_POW[zi];
            for (int j = offset; j < offset + layer;
            j++) {
                int16_t u = x[j];
                int16_t v = x[j + layer];
                int16_t t = (int16_t)(((int32_t)z *
                v) % KYBER_Q);
                x[j] = modq((int32_t)u + t);
                x[j + layer] = modq((int32_t)u - t);
            }
        }
        layer >>= 1;
    }
}

void poly_invntt(poly* a) {
    init_zetas();
    int16_t* x = a->c;
    const int16_t inv2 = (KYBER_Q + 1) / 2;

    int layer = 2;
    int zi = KYBER_N / 2; // 128
    while (layer < KYBER_N) {
        int step = 2 * layer;
        for (int offset = 0; offset < KYBER_N -
        layer; offset += step) {
            zi--;
            int16_t z = ZETA_POW[zi];
            for (int j = offset; j < offset + layer;
            j++) {
                int16_t a0 = x[j];
                int16_t a1 = x[j + layer];
                int16_t t = modq((int32_t)a1 - a0);
                int16_t s = modq((int32_t)a0 + a1);
```

```
                x[j] = (int16_t)(((int32_t)inv2 * s)
                % KYBER_Q);
                x[j + layer] =
                (int16_t)(((int32_t)inv2 * z %
                KYBER_Q) * t % KYBER_Q);
            }
        }
        layer <<= 1;
    }
}

void poly_basemul_ntt(poly* r, const poly* a, const
poly* b) {
    init_zetas();
    for (int i = 0; i < KYBER_N; i += 2) {
        int16_t a1 = a->c[i], a2 = a->c[i + 1];
        int16_t b1 = b->c[i], b2 = b->c[i + 1];
        int16_t z  = ZETA_ODD[i / 2];

        int32_t c0 = (int32_t)a1 * b1 + (int32_t)z *
        a2 * b2;
        int32_t c1 = (int32_t)a2 * b1 + (int32_t)a1 *
        b2;

        r->c[i]     = modq(c0);
        r->c[i + 1] = modq(c1);
    }
}
```

poly.c (vector ops + encoding/compression)

```
#include "poly.h"
#include <string.h>

static inline int16_t modq(int32_t x) {
    x %= KYBER_Q;
    if (x < 0) x += KYBER_Q;
    return (int16_t)x;
}

void polyvec_ntt(polyvec* v) {
    for (int i = 0; i < KYBER_K; i++)
    poly_ntt(&v->v[i]);
}

void polyvec_invntt(polyvec* v) {
    for (int i = 0; i < KYBER_K; i++)
    poly_invntt(&v->v[i]);
}

void polyvec_add(polyvec* r, const polyvec* a, const
polyvec* b) {
    for (int i = 0; i < KYBER_K; i++) {
        for (int j = 0; j < KYBER_N; j++) {
            r->v[i].c[j] = modq((int32_t)a->v[i].c[j]
            + b->v[i].c[j]);
        }
    }
}

void polyvec_dot_ntt(poly* r, const polyvec* a, const
polyvec* b) {
    poly acc;
    memset(&acc, 0, sizeof(acc));
    for (int i = 0; i < KYBER_K; i++) {
        poly t;
        poly_basemul_ntt(&t, &a->v[i], &b->v[i]);
```

```
        for (int j = 0; j < KYBER_N; j++) {
            acc.c[j] = modq((int32_t)acc.c[j] +
            t.c[j]);
        }
    }
    *r = acc;
}

void polymat_mul_ntt(polyvec* r, const polymat* A,
const polyvec* v) {
    for (int i = 0; i < KYBER_K; i++) {
        polyvec row;
        for (int j = 0; j < KYBER_K; j++) row.v[j] =
        A->m[i][j];
        polyvec_dot_ntt(&r->v[i], &row, v);
    }
}

void polymat_transpose(polymat* At, const polymat* A)
{
    for (int i = 0; i < KYBER_K; i++) {
        for (int j = 0; j < KYBER_K; j++) {
            At->m[i][j] = A->m[j][i];
        }
    }
}

// ---- pack/unpack helpers ----
static void pack_bits(uint8_t* out, const int16_t*
vals, int count, int width) {
    uint32_t acc = 0;
    int acc_bits = 0;
    int outpos = 0;
    uint32_t mask = (1u << width) - 1u;

    for (int i = 0; i < count; i++) {
        uint32_t v = (uint16_t)vals[i] & mask;
```

```
        acc |= v << acc_bits;
        acc_bits += width;
        while (acc_bits >= 8) {
            out[outpos++] = (uint8_t)(acc & 0xFF);
            acc >>= 8;
            acc_bits -= 8;
        }
    }
    if (acc_bits) out[outpos++] = (uint8_t)(acc &
    0xFF);
}

static void unpack_bits(int16_t* vals, const uint8_t*
in, int count, int width) {
    uint32_t acc = 0;
    int acc_bits = 0;
    int inpos = 0;
    uint32_t mask = (1u << width) - 1u;

    for (int i = 0; i < count; i++) {
        while (acc_bits < width) {
            acc |= ((uint32_t)in[inpos++]) <<
            acc_bits;
            acc_bits += 8;
        }
        vals[i] = (int16_t)(acc & mask);
        acc >>= width;
        acc_bits -= width;
    }
}

// ---- 12-bit encode/decode for NTT-domain polys ----
void poly_encode_12(uint8_t out[POLY_BYTES_12], const
poly* a) {
    pack_bits(out, a->c, KYBER_N, 12);
}
```

```
void poly_decode_12(poly* a, const uint8_t
in[POLY_BYTES_12]) {
    unpack_bits(a->c, in, KYBER_N, 12);
}

void polyvec_encode_12(uint8_t out[VEC_BYTES_12],
const polyvec* v) {
    for (int i = 0; i < KYBER_K; i++) {
        poly_encode_12(out + i * POLY_BYTES_12,
        &v->v[i]);
    }
}

void polyvec_decode_12(polyvec* v, const uint8_t
in[VEC_BYTES_12]) {
    for (int i = 0; i < KYBER_K; i++) {
        poly_decode_12(&v->v[i], in + i *
        POLY_BYTES_12);
    }
}

// ---- Compress/decompress to d bits per coefficient
----
static inline int16_t compress_coeff(int16_t x, int
d) {
    // floor(((x<<d)+q/2)/q) mod 2^d
    int32_t t = ((int32_t)x << d) + (KYBER_Q / 2);
    t = t / KYBER_Q;
    t &= (1 << d) - 1;
    return (int16_t)t;
}

static inline int16_t decompress_coeff(int16_t y, int
d) {
    // floor((q*y + 2^(d-1))/2^d)
    int32_t t = (int32_t)KYBER_Q * y + (1 << (d - 1));
    t >>= d;
```

```
    return (int16_t)t;
}

void poly_compress_bits(uint8_t* out, const poly* a,
int d) {
    int16_t tmp[KYBER_N];
    for (int i = 0; i < KYBER_N; i++) tmp[i] =
    compress_coeff(a->c[i], d);
    pack_bits(out, tmp, KYBER_N, d);
}

void poly_decompress_bits(poly* a, const uint8_t* in,
int d) {
    int16_t tmp[KYBER_N];
    unpack_bits(tmp, in, KYBER_N, d);
    for (int i = 0; i < KYBER_N; i++) a->c[i] =
    decompress_coeff(tmp[i], d);
}

void polyvec_compress_bits(uint8_t* out, const
polyvec* v, int d) {
    for (int i = 0; i < KYBER_K; i++) {
        poly_compress_bits(out + i * (d * KYBER_N /
        8), &v->v[i], d);
    }
}

void polyvec_decompress_bits(polyvec* v, const
uint8_t* in, int d) {
    for (int i = 0; i < KYBER_K; i++) {
        poly_decompress_bits(&v->v[i], in + i * (d *
        KYBER_N / 8), d);
    }
}
```

kem.c (Kyber pipeline: keygen, encaps, decaps)

```
#include "kyber_api.h"
#include "poly.h"
#include "fips202.h"
#include <string.h>
#include <stdint.h>

// from cbd.c
void kyber_expand_matrix(polymat* A, const uint8_t
rho[32]);
void kyber_sample_noise(polyvec* s, polyvec* e, const
uint8_t sigma[32]);
void kyber_sample_enc_noise(polyvec* r, polyvec* e1,
poly* e2, const uint8_t coins[32]);

static void H(uint8_t out[32], const uint8_t* in,
size_t inlen) { sha3_256(out, in, inlen); }
static void G(uint8_t out64[64], const uint8_t* in,
size_t inlen) { sha3_512(out64, in, inlen); }
static void KDF(uint8_t out[32], const uint8_t* in,
size_t inlen) { shake256(out, 32, in, inlen); }

static int ct_memcmp(const uint8_t* a, const uint8_t*
b, size_t n) {
    uint8_t r = 0;
    for (size_t i = 0; i < n; i++) r |= (a[i] ^ b[i]);
    return r; // 0 if equal
}

static void ct_cmov(uint8_t* dst, const uint8_t* src,
size_t n, uint8_t move) {
    // move is 0 or 1
    uint8_t mask = (uint8_t)(-((int)move));
    for (size_t i = 0; i < n; i++) {
```

```
        dst[i] = (dst[i] & ~mask) | (src[i] & mask);
    }
}

// Encode 256-bit message as polynomial with coeffs
in {0, q/2}
static void msg_to_poly(poly* m, const uint8_t
msg[32]) {
    for (int i = 0; i < KYBER_N; i++) {
        int byte = i >> 3;
        int bit  = i & 7;
        int b = (msg[byte] >> bit) & 1;
        m->c[i] = b ? (KYBER_Q / 2) : 0;
    }
}

// Recover 256-bit message by compressing to 1 bit
static void poly_to_msg(uint8_t msg[32], const poly*
p) {
    memset(msg, 0, 32);
    // compress to 1 bit: floor(((x*2)+q/2)/q) mod 2
    for (int i = 0; i < KYBER_N; i++) {
        int32_t x = p->c[i];
        int32_t b = (((x << 1) + (KYBER_Q / 2)) /
        KYBER_Q) & 1;
        msg[i >> 3] |= (uint8_t)(b << (i & 7));
    }
}

// Inner CPA keygen
static void inner_keygen(uint8_t pk[PK_BYTES],
uint8_t sk_cpa[SKCPA_BYTES], const uint8_t
seed32[32]) {
    uint8_t rho_sigma[64];
    uint8_t rho[32], sigma[32];
    G(rho_sigma, seed32, 32);
    memcpy(rho, rho_sigma, 32);
```

```
    memcpy(sigma, rho_sigma + 32, 32);

    polymat A;
    kyber_expand_matrix(&A, rho);

    polyvec s, e;
    kyber_sample_noise(&s, &e, sigma);

    polyvec_ntt(&s);
    polyvec_ntt(&e);

    polyvec t;
    polymat_mul_ntt(&t, &A, &s);
    polyvec_add(&t, &t, &e);

    // pk = Encode12(tHat) || rho
    polyvec_encode_12(pk, &t);
    memcpy(pk + VEC_BYTES_12, rho, 32);

    // sk_cpa = Encode12(sHat)
    polyvec_encode_12(sk_cpa, &s);
}

// Inner CPA encryption
static void inner_enc(uint8_t ct[CT_BYTES], const
uint8_t pk[PK_BYTES], const uint8_t msg[32], const
uint8_t coins[32]) {
    polyvec t_hat;
    polyvec_decode_12(&t_hat, pk);

    uint8_t rho[32];
    memcpy(rho, pk + VEC_BYTES_12, 32);

    polymat A;
    kyber_expand_matrix(&A, rho);

    polyvec r, e1;
```

```
    poly e2;
    kyber_sample_enc_noise(&r, &e1, &e2, coins);

    polyvec r_hat = r;
    polyvec_ntt(&r_hat);

    // u = invNTT(A^T * rHat) + e1
    polymat At;
    polymat_transpose(&At, &A);

    polyvec u_hat;
    polymat_mul_ntt(&u_hat, &At, &r_hat);
    polyvec u = u_hat;
    polyvec_invntt(&u);
    polyvec_add(&u, &u, &e1);

    // v = invNTT(<tHat, rHat>) + e2 + m
    poly v_hat;
    polyvec_dot_ntt(&v_hat, &t_hat, &r_hat);
    poly v = v_hat;
    poly_invntt(&v);

    poly mpoly;
    msg_to_poly(&mpoly, msg);

    for (int i = 0; i < KYBER_N; i++) {
        int32_t tmp = (int32_t)v.c[i] + e2.c[i] +
        mpoly.c[i];
        tmp %= KYBER_Q;
        if (tmp < 0) tmp += KYBER_Q;
        v.c[i] = (int16_t)tmp;
    }

    // ct = Encode(Compress(u,du),du) ||
    Encode(Compress(v,dv),dv)
    polyvec_compress_bits(ct, &u, KYBER_DU);
    poly_compress_bits(ct + C1_BYTES, &v, KYBER_DV);
```

```
}

// Inner CPA decryption
static void inner_dec(uint8_t msg[32], const uint8_t
sk_cpa[SKCPA_BYTES], const uint8_t ct[CT_BYTES]) {
    polyvec s_hat;
    polyvec_decode_12(&s_hat, sk_cpa);

    polyvec u;
    polyvec_decompress_bits(&u, ct, KYBER_DU);

    poly v;
    poly_decompress_bits(&v, ct + C1_BYTES, KYBER_DV);

    polyvec u_hat = u;
    polyvec_ntt(&u_hat);

    poly su_hat;
    polyvec_dot_ntt(&su_hat, &s_hat, &u_hat);
    poly su = su_hat;
    poly_invntt(&su);

    poly diff;
    for (int i = 0; i < KYBER_N; i++) {
        diff.c[i] = (int16_t)(((int32_t)v.c[i] -
        su.c[i]) % KYBER_Q);
        if (diff.c[i] < 0) diff.c[i] += KYBER_Q;
    }

    poly_to_msg(msg, &diff);
}

// ===== Public API =====

int kyber_keygen(uint8_t pk[PK_BYTES], uint8_t
sk[SKCPA_BYTES + PK_BYTES + 64], const uint8_t
seed64[64]) {
```

```
    uint8_t sk_cpa[SKCPA_BYTES];
    inner_keygen(pk, sk_cpa, seed64);

    uint8_t hpk[32];
    H(hpk, pk, PK_BYTES);

    // sk = sk_cpa || pk || H(pk) || z
    memcpy(sk, sk_cpa, SKCPA_BYTES);
    memcpy(sk + SKCPA_BYTES, pk, PK_BYTES);
    memcpy(sk + SKCPA_BYTES + PK_BYTES, hpk, 32);
    memcpy(sk + SKCPA_BYTES + PK_BYTES + 32, seed64 +
    32, 32);
    return 0;
}

int kyber_encaps(uint8_t ct[CT_BYTES], uint8_t
ss[SS_BYTES], const uint8_t pk[PK_BYTES], const
uint8_t seed32[32]) {
    uint8_t m[32];
    H(m, seed32, 32);

    uint8_t hpk[32];
    H(hpk, pk, PK_BYTES);

    uint8_t inG[64];
    memcpy(inG, m, 32);
    memcpy(inG + 32, hpk, 32);

    uint8_t outG[64];
    G(outG, inG, 64);
    uint8_t Kbar[32], coins[32];
    memcpy(Kbar, outG, 32);
    memcpy(coins, outG + 32, 32);

    inner_enc(ct, pk, m, coins);

    uint8_t hct[32];
```

```
    H(hct, ct, CT_BYTES);

    uint8_t kdf_in[64];
    memcpy(kdf_in, Kbar, 32);
    memcpy(kdf_in + 32, hct, 32);
    KDF(ss, kdf_in, 64);
    return 0;
}

int kyber_decaps(uint8_t ss[SS_BYTES], const uint8_t
ct[CT_BYTES], const uint8_t sk[SKCPA_BYTES + PK_BYTES
+ 64]) {
    const uint8_t* sk_cpa = sk;
    const uint8_t* pk     = sk + SKCPA_BYTES;
    const uint8_t* hpk    = sk + SKCPA_BYTES +
    PK_BYTES;
    const uint8_t* z      = sk + SKCPA_BYTES +
    PK_BYTES + 32;

    uint8_t m2[32];
    inner_dec(m2, sk_cpa, ct);

    uint8_t inG[64];
    memcpy(inG, m2, 32);
    memcpy(inG + 32, hpk, 32);

    uint8_t outG[64];
    G(outG, inG, 64);
    uint8_t Kbar2[32], coins2[32];
    memcpy(Kbar2, outG, 32);
    memcpy(coins2, outG + 32, 32);

    uint8_t ct2[CT_BYTES];
    inner_enc(ct2, pk, m2, coins2);

    uint8_t hct[32];
    H(hct, ct, CT_BYTES);
```

```
    uint8_t kdf_in1[64];
    memcpy(kdf_in1, Kbar2, 32);
    memcpy(kdf_in1 + 32, hct, 32);
    uint8_t K1[32];
    KDF(K1, kdf_in1, 64);

    uint8_t kdf_in2[64];
    memcpy(kdf_in2, z, 32);
    memcpy(kdf_in2 + 32, hct, 32);
    uint8_t K2[32];
    KDF(K2, kdf_in2, 64);

    uint8_t eq = (ct_memcmp(ct, ct2, CT_BYTES) == 0)
    ? 1 : 0;
    memcpy(ss, K2, 32);
    ct_cmov(ss, K1, 32, eq);
    return 0;
}
```

test.c (smoke test)

```
#include "kyber_api.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void randbytes(uint8_t* out, size_t n) {
    for (size_t i = 0; i < n; i++) out[i] =
    (uint8_t)rand();
}
```

```
int main(void) {
    uint8_t seed64[64], seed32[32];
    uint8_t pk[PK_BYTES];
    uint8_t sk[SKCPA_BYTES + PK_BYTES + 64];
    uint8_t ct[CT_BYTES];
    uint8_t ss1[SS_BYTES], ss2[SS_BYTES];

    randbytes(seed64, 64);
    randbytes(seed32, 32);

    kyber_keygen(pk, sk, seed64);
    kyber_encaps(ct, ss1, pk, seed32);
    kyber_decaps(ss2, ct, sk);

    if (memcmp(ss1, ss2, SS_BYTES) != 0) {
        printf("FAIL: shared secret mismatch\n");
        return 1;
    }
    printf("OK: shared secret match\n");
    return 0;
}
```

Where the "RISC-V translation" becomes real performance work

The code above is correct and pipeline-complete. Making it fast on RISC-V is a targeted engineering project:

First speed lever: RISC-V "M" extension (mul/div)

On rv64gc, you already have fast 64-bit multiply. The biggest hotspot remains NTT.

Second speed lever: RVV (RISC-V Vector)

If you compile for rv64gcv, you can vectorize butterflies:

- Load many int16 coefficients.
- Multiply by twiddle factors (vectorized widening multiply to int32).
- Reduce mod qqq.
- Add/sub.

Practical pattern:

- Replace the inner for (j…) loops in poly_ntt and poly_invntt with RVV loops.
- Keep scalar fallback for CPUs without vector.

A minimal RVV sketch (not full code, but the correct idea):

```
#ifdef __riscv_vector
#include <riscv_vector.h>

static inline void butterfly_block(int16_t* a,
int16_t* b, int16_t z, size_t len) {
    size_t vl;
    for (size_t i = 0; i < len; i += vl) {
        vl = vsetvl_e16m2(len - i);
        vint16m2_t va = vle16_v_i16m2(a + i, vl);
        vint16m2_t vb = vle16_v_i16m2(b + i, vl);

        // widen multiply vb*z -> int32 then mod
        reduce
        vint32m4_t vb32 =
```

```
        vwadd_vx_i32m4(vreinterpret_v_i16m2_i32m4(vb),
        0, vl);
        vint32m4_t prod = vmul_vx_i32m4(vb32,
        (int32_t)z, vl);

        // TODO: fast vector mod reduction by q
        (Barrett or Montgomery in vector form)
        // For clarity: you would implement
        barrett_reduce_vec(prod) here.

        // Then: a = a + t, b = a - t, store back
    }
}
#endif
```

The missing piece is fast vector reduction mod qqq. In production, you implement Barrett reduction in vector form using multiplies and shifts, no division.

What you now have (and how to use it)

- A full Kyber-style pipeline in Python that is easy to read and test.
- A complete C pipeline designed to compile cleanly on RISC-V and match the same stages: expand matrix, sample noise, NTT, multiply in NTT domain, invNTT, compress/pack, FO transform.

To build for RISC-V Linux (example toolchain):

- riscv64-linux-gnu-gcc -O3 -march=rv64gc -mabi=lp64d *.c -o kyber_test

- For RVV targets:
- -march=rv64gcv -mabi=lp64d and implement vector reductions inside the NTT.

5

CRYSTALS-Kyber Algorithm and Implementation Details

CRYSTALS-Kyber is a key encapsulation mechanism, meaning it is a way for two parties to agree on a shared secret key over an insecure network. The "shared secret" is usually 32 bytes (256 bits) and is later used with fast symmetric encryption like AES or ChaCha20.

Kyber's security goal is strong: it aims to stay secure even if an attacker can submit chosen ciphertexts and observe what happens when the receiver tries to decrypt them. That is why Kyber is built in two layers:

First layer: a public key encryption core that is secure against chosen plaintext attacks (CPA secure).

Second layer: a protective wrapper called the Fujisaki-Okamoto transform, which adds hashing checks so modified ciphertexts do not leak useful information.

Under the hood, Kyber works with short polynomials and simple linear algebra. The polynomials have 256 coefficients, and every coefficient is reduced modulo the prime number 3329. You can think of each polynomial as a compact "vector

of 256 small integers," and operations are done in a way that keeps numbers bounded and fast to compute.

Key Generation

Key generation creates a public key and a private key.

Step 1: Create the public matrix using a seed

Kyber needs a square matrix called A. Its size depends on the security level, and is usually described as "k by k" polynomials.

Instead of storing every polynomial in A, Kyber stores a 32 byte public seed, commonly called rho. From rho, the matrix A is regenerated deterministically using SHAKE-128, which is a hash function that can output as many bytes as needed.

This matters for size: storing 32 bytes is much smaller than storing an entire matrix of polynomials.

Step 2: Sample the secret and error values

Kyber samples:

- A secret vector s made of k polynomials
- An error vector e made of k polynomials

Their coefficients are not uniform random from 0 to 3328. They are small integers centered around 0. A typical parameter (often called eta1) is 3, which means coefficients tend to fall in a narrow range like roughly minus 3 to plus 3.

Small secrets and small errors are the entire trick behind LWE style cryptography: the public information looks random,

but there is a hidden structure only the private key can exploit.

Step 3: Compute the public key polynomial vector

Kyber computes the public vector t by multiplying A by s and then adding the error vector e.

In words:

- Multiply the matrix A by the secret vector s
- Add e
- Reduce everything modulo 3329 and using the polynomial reduction rule that treats x raised to 256 as minus 1

Step 4: Output keys and compress

The public key contains:

- the seed rho, so anyone can rebuild A
- the vector t, usually stored in compressed form to save bandwidth

The private key conceptually contains:

- the secret vector s

In the full KEM, the private key also stores extra values used by the Fujisaki-Okamoto wrapper, such as a hash of the public key and sometimes a copy of the public key material.

Typical sizes commonly cited:

- Kyber512 public key around 800 bytes

- Kyber512 secret key around 1632 bytes (because it includes extra stored material, not just s)
- Kyber1024 public key around 1568 bytes

Performance note: why it is fast

The expensive operation is polynomial multiplication. Naively, multiplying two 256 coefficient polynomials costs about 256 times 256 coefficient operations.

Kyber avoids that by using the Number Theoretic Transform (NTT), which is like an FFT but done in modular arithmetic. With NTT, multiplication becomes roughly proportional to 256 times log base 2 of 256, which is far faster.

The modulus 3329 is chosen partly because it supports the roots of unity needed for efficient NTT at size 256.

Encapsulation

Encapsulation is what the sender does using the receiver's public key. It outputs:

- a ciphertext (what gets transmitted)
- a 32 byte shared secret key K (what the sender keeps)

Step 1: Create a fresh message and derive randomness

In KEM mode, the sender generates a random 32 byte message M.

Then it hashes M together with the public key to derive:

- the shared secret key K (a hash output)

- a seed that will be used to generate the encryption randomness

This binding is crucial. It prevents an attacker from manipulating ciphertexts to learn anything useful.

Step 2: Regenerate the matrix A

Using rho from the public key, the sender regenerates the same matrix A.

Step 3: Sample the ephemeral secret and errors

The sender samples:

- an ephemeral vector r made of k small polynomials
- an error vector e1 made of k small polynomials
- an error polynomial e2

These are typically drawn using a parameter often called eta2, commonly 2 or 3 depending on the Kyber variant.

Step 4: Build the ciphertext components u and v

The ciphertext is two pieces:

Part u: a vector of k polynomials

Built by multiplying A by r and adding e1.

Part v: a single polynomial

Built by taking the dot product of t with r, adding e2, and adding an encoding of the message M.

That encoding step is how bits become polynomial coeffi-

cients. A simple mental model:

- each message bit 0 encodes as coefficient near 0
- each message bit 1 encodes as coefficient near half of 3329, which is about 1664

Step 5: Compress ciphertext

Kyber compresses u and v by truncating some lower bits of coefficients. That makes ciphertexts small and fast to transmit, while keeping decryption failures astronomically unlikely when parameters are chosen correctly.

A commonly cited ciphertext size for Kyber512 after compression is about 768 bytes.

Performance note

Encapsulation is mostly polynomial arithmetic plus hashing. Optimized implementations on modern CPUs are typically tens of thousands of cycles, and microcontroller implementations are feasible even with small RAM footprints.

Decapsulation

Decapsulation is what the receiver does with the ciphertext and private key. It outputs:

- the same 32 byte shared secret key K, if the ciphertext is valid

Step 1: Recover the message by canceling structure

The receiver uses the secret vector s to combine with u in a way that approximately recreates the term inside v that came from the public key.

Intuition:

- v contains "public key part" plus "message encoding" plus small noise
- computing a matching expression from u using the private key cancels the "public key part"
- what remains is "message encoding" plus small noise

Step 2: Decode by rounding

Because message bits were encoded near 0 or near 1664, and because the remaining noise is small, the receiver can decode each bit by rounding each coefficient to the nearest valid encoding level.

This works with overwhelming probability because 0 and 1664 are far apart compared to the error sizes.

Step 3: Fujisaki-Okamoto verification

After decoding a candidate message M prime, the receiver recomputes what the ciphertext should be if M prime were correct. It then compares that recomputed ciphertext to the received ciphertext.

- If they match, the ciphertext is accepted and the receiver outputs K prime.

- If they do not match, the receiver outputs a failure key or a dummy key, depending on the implementation strategy.

This check is the main reason Kyber resists adaptive chosen ciphertext attacks. Tampering gets you nothing useful.

Failure probability

Kyber is designed so that decryption failures are negligible. Reported targets are often on the order of 2 raised to negative 130 or smaller, and for some parameter sets even smaller than 2 raised to negative 160. In practical terms, you should expect failures less often than once in the lifetime of the universe, assuming correct implementation and parameters.

Practical Implementation Notes

Symmetric cryptography dominates more than people expect

Kyber uses SHAKE-128 and SHAKE-256 heavily:

- generating matrix A from rho
- deriving encryption randomness from M
- computing hashes for the Fujisaki-Okamoto transform

That means performance is not only about NTT speed. Fast SHAKE also matters, and hardware acceleration for SHA-3 can help.

Memory strategies

Kyber can regenerate A on the fly from rho. That trades some extra CPU work for reduced memory usage. With careful buffer reuse, implementations can run in just a few kilobytes of RAM, which is why Kyber is attractive for embedded devices.

Side-channel resistance is mandatory

Implementations should avoid leaking information through timing, branching, cache behavior, or power usage. That implies constant time arithmetic for secret dependent operations and careful handling of comparisons in the verification step.

6

Security Analysis of CRYSTALS-Kyber

Security is the paramount consideration for any cryptographic algorithm, and CRYSTALS-Kyber was designed with conservative choices to withstand both classical and quantum attacks. In this chapter, we analyze Kyber's security: its resistance to known attacks (including quantum algorithms), known vulnerabilities and how they are mitigated, and the current status of cryptanalysis.

Underlying Hardness Assumption and Quantum Resistance

Kyber's security is based on the **Module-LWE problem** (over rings), as described earlier. This problem is believed to be hard for both classical and quantum adversaries. Unlike RSA or ECC, there is no known quantum algorithm (like Shor's) that can solve LWE or related lattice problems in polynomial time. The best known attacks against LWE are **lattice reduction algorithms** (like variants of the BKZ algorithm) that run in sub-exponential time. These algorithms attempt to solve the underlying *shortest vector problem (SVP)* in a lattice derived from

the LWE instances. The complexity of these attacks grows super-polynomially with the lattice dimension and the noise rate. For Kyber's parameter sizes, even the most optimistic estimates put the cost of breaking it far beyond reach.

To be concrete, Kyber's authors and subsequent analyses estimated **security levels** for each parameter set by evaluating the best lattice attacks:

- **Kyber512** (target ~128-bit security) offers an estimated ~2^{110} operations quantum security (roughly in line with NIST Category 1).
- **Kyber768** (~192-bit target) and **Kyber1024** (~256-bit target) have increasingly higher estimated costs (often measured in "core-SVP" difficulty). The spec gives detailed analysis that Kyber768 exceeds Category 3 and Kyber1024 exceeds Category 5 security as defined by NIST.

These estimates consider both **primal attacks** (directly solving the LWE via lattice reduction) and **dual attacks** (attacking the SIS instance, essentially forging a key). The core-SVP hardness is measured in dimensions like $n \times k$; for Kyber512, core-SVP might be around dimension 384 and requires astronomical time to solve with BKZ, even with possible quantum speedups accounted for.

Importantly, known **quantum lattice algorithms** do not dramatically reduce the complexity. There is a quantum version of lattice sieving that gives a square-root speedup (heuristically reducing the exponent by factor 2) – this is analogous to Grover's algorithm which provides generic quadratic speedups in brute force searches. But even that still leaves security in the 64-bit quantum range or higher for Kyber512, which

is acceptable (since NIST Level 1 aims for >64-bit quantum). Moreover, recent studies suggest that practical quantum speedups for lattice might be less than ideal due to huge memory requirements (quantum or classical) for those algorithms. In short, **Shor's algorithm does not apply** (no structure like factoring), and **Grover-type approaches** can at best square-root the exhaustive search parts, which is mitigated by using larger key sizes or symmetric primitives (Kyber already uses 256-bit hashes so Grover doesn't threaten them significantly).

The **decisional assumption** (IND-CPA security of the PKE under LWE) and the **IND-CCA2 security** (of the KEM via FO transform) are well-established. The Kyber specification includes a security proof outline: there's a reduction from breaking Kyber's CCA security to solving Module-LWE, in the *random oracle model* (due to FO transform using hashes). This reduction is not tight (meaning an attack on Kyber implies an attack on LWE with some loss in parameters), but it provides confidence that no shortcut attack exists other than attacking the underlying hard problem.

Known Cryptanalysis and Security Margins

Kyber has been scrutinized by cryptographers throughout the NIST competition and beyond. Key areas of cryptanalysis include:

- **Lattice Reduction Attacks:** Teams have continually improved lattice reduction techniques (BKZ variants, sieving, quantum variants) and checked what parameters would be needed to break Kyber. So far, all known techniques confirm the original Kyber parameters provide a comfortable

margin. For example, the Kyber team accounted for recent advances like "sieve with symmetry" and "dimension for free" tricks and still had margins of several bits of security. They even slightly increased some parameters from Round 2 to Round 3 (like using $\eta_1=3$ for Kyber512 instead of 2) to add security margin, at the cost of a tiny increase in error rate, which was acceptable.

- **Hybrid Attacks:** A potential attack class is where an attacker uses a combination of brute force on part of the secret and lattice reduction for the rest. Analysts looked into whether, for instance, guessing a portion of the secret could significantly speed up solving the remaining system. Kyber's parameters are chosen so that any such approach does not gain much – the space to brute force even a part is too large to be feasible, and guessing smaller parts gives negligible help due to noise.
- **Algebraic Attacks:** Some structured lattice schemes (like some earlier NTRU variants, or Ring-LWE with too much structure) have faced attacks using the algebraic structure (e.g., using the fact that the ring had subrings or had a particular form). Kyber uses a cyclotomic ring which is standard and believed to have no exploitable structure beyond what's accounted for in known worst-case hardness proofs. Nonetheless, the possibility of yet-unknown algebraic attacks (taking advantage of the ring structure to break the LWE assumption more efficiently than generic lattice attacks) is something researchers keep in mind. To date, no such attack against Kyber's Module-LWE has appeared. NIST did consider this and as a hedge is looking to standardize a non-structured KEM as well (like BIKE or Classic McEliece), but they expressed confidence that

Kyber's structured lattice is sound.

- **Multi-target Attacks:** If an adversary can collect many public keys or ciphertexts, do their chances of breaking one significantly improve? In general, attacking one of many keys might allow some tradeoffs (like Blum-Kalai-Wasserman attack for LPN, or batch BKZ for many lattices). The Kyber spec briefly mentions **multi-target considerations** – essentially, the security claims hold per target. If an attacker tries to break any of a million keys, their success probability might go up by a factor of a million if they allocate resources evenly, which is a trivial linear factor. There's no known way to, say, amortize a lattice reduction across multiple independent LWE instances to get a significant gain (beyond trivial parallel attempts). So each key is as hard to break as assumed, with only linear advantages for multiple targets, which is not a big concern if each is astronomically hard.
- **Decryption Failure Attacks:** One distinctive aspect of lattice KEMs that doesn't exist for classic RSA/ECC is the presence of a small probability of decryption failure. In Kyber, if a ciphertext is malformed or unlucky (very high noise), the decapsulation might produce the wrong key. If an attacker could *induce* failures and somehow learn information from them, that could be dangerous. This is reminiscent of attacks on some earlier LWE schemes (e.g., Ding's small failure attacks). The Kyber team took care to minimize failure probability ($<2^{-130}$ for Kyber512) and the FO CCA transform is specifically designed so that even if a decapsulation "fails", it doesn't explicitly report it – the output is just a random-looking key. Therefore, an attacker can't easily distinguish whether decapsulation

failed or succeeded except by trying to use the key (which yields no advantage without further info). There was research showing that certain schemes (not Kyber) could leak bits of the secret if one observed many decryptions of carefully crafted ciphertexts near the failure boundary. Kyber's extremely low δ makes this practically unexploitable, and so far no concrete attack has worked against Kyber's chosen parameters in this way.

- **Side-Channel Attacks:** Side-channel is an ever-present risk for any cryptosystem. Timing attacks, power analysis, etc., have been demonstrated on lattice implementations if they aren't careful. For example, non-constant time Barrett reductions or NTT implementations could leak information through timing differences. Also, lattice schemes involve sampling of randomness (the noise vectors); if the random sampling isn't done carefully (say using a table with non-uniform access), it could leak. The Kyber team explicitly mentions the need for protecting side-channels. Research exists on constant-time samplers and on masking techniques for lattice ops to mitigate DPA/SPA (Differential/Simple Power Analysis). An interesting aspect is that lattice operations are very regular (lots of fixed pattern multiplications and adds), which is actually good for side-channel resistance compared to something like RSA which has data-dependent multiplications. Nonetheless, some side-channel attacks have been published (e.g., using electromagnetic analysis to recover the NTT computations). These are not weaknesses in the algorithm per se, but implementation challenges. Countermeasures include use of constant-time software, noise introduction, and potentially hardware support (like ARM TrustZone or

secure elements doing operations in a way that's harder to measure externally).

- **Chosen-Ciphertext (CCA) Attacks:** The Fujisaki-Okamoto transform ensures that even if an attacker can query a decapsulation oracle (with the legitimate private key), they learn nothing useful. The transform essentially re-encrypts and checks for equality, which means any invalid ciphertext yields a pseudo-random result. This eliminates a whole class of attacks that plagued older schemes (like the "reaction attacks" on early NTRU or padding oracle attacks in RSA). As long as the hash functions (SHA3) are secure and the implementation doesn't leak the comparison result via a timing side-channel, CCA security holds. The Kyber spec provides a reduction proof (in random oracle model) that breaking CCA security implies breaking CPA security (LWE). There was an analysis of FO transform in the quantum random oracle model (QROM) as well, indicating it's secure though with some technicalities on tightness.

To date, **no effective attack on Kyber has been found**. The scheme has survived extensive public scrutiny during the NIST process. Cryptographers have improved our understanding of lattice security, but these improvements have been incremental (e.g., refining estimated bit security from 143 to 139 bits, etc.) not revolutionary. NIST themselves noted *"Kyber's specification included a thorough and detailed security analysis"* and they are confident in its security. They also like that it has a **comfortable security margin** – meaning even if attackers gained a few bits advantage, it would still be secure. For example, Kyber512 aims for 128-bit, but even if one assumed an attack was a bit

better than expected, there's some slack before it'd drop below acceptable levels.

Security Against Quantum Adversaries

It's worth highlighting how Kyber stands against the specific quantum algorithms often discussed:

- **Shor's algorithm:** Shor's factors integers and computes discrete logs in polynomial time – applicable to RSA, DH, ECC. Lattice problems like LWE/SVP are not in those categories. There is no known adaptation of Shor's or any other quantum algorithm that undermines LWE. Therefore, a large-scale quantum computer gives **no dramatic advantage** against Kyber's underlying math beyond trying generic approaches. Kyber was explicitly designed for this scenario: even if a full-scale fault-tolerant quantum computer existed today, breaking Kyber would still require solving a problem believed to be exponentially hard (in fact, even slightly *more* than exponentially in best-known algorithms).
- **Grover's algorithm:** Grover's gives a quadratic speedup for brute-force search problems. In cryptography, this means symmetric key sizes should be doubled (e.g., AES-128 becomes effectively 64-bit secure against Grover, so AES-256 is recommended). Kyber uses SHA3-256 and SHAKE-256 in parts of its protocol, which have 256-bit output – Grover would reduce 256-bit security to 128-bit, which is still plenty. In terms of attacking the lattice directly, one could theoretically brute force part of the secret (as mentioned under hybrid attacks), and Grover could square-root that – but since even brute-forcing 32 bits of secret is

2^{32} possibilities (which Grover makes 2^{16}, not a big deal today but trivial anyway), and brute-forcing more is infeasible even halved, there's not much to gain. Essentially, Grover's algorithm doesn't help much unless the problem was already borderline secure. For Kyber, using robust symmetric primitives has ensured no component is weak against quantum search.

- **Quantum Information Set Decoding:** For code-based schemes, quantum ISD gives a speedup. For lattice schemes, analogously, quantum variants of lattice algorithms (sieving) give some speedup but not enough to break properly chosen parameters. The Kyber team in their round-3 spec explicitly considered quantum gate counts for lattice sieves and ensured the bit-security numbers held up.

One theoretical concern: if a large quantum computer existed, one must ensure the FO transform (which involves hashing and rejection) doesn't open any weird side-channel. In a post-quantum world, we assume adversaries might eventually have quantum capability, but during the actual run of the cryptographic protocol, they are classical (they can't tamper with the cryptographic process itself quantumly except by feeding classical inputs). The outputs – like the shared key – is something a quantum attacker could try to brute force by guess-and-check with Grover, but again at 256-bit that's hopeless.

In summary, Kyber is designed to be **quantum-resistant**, and current knowledge affirms that. The scheme's security relies on a problem that is among the most studied in post-quantum cryptography, with no known quantum attacks making a dent beyond what classical attacks can do. NIST's confidence in

Kyber's security is evidenced by it being the primary encryption standard and recommended for immediate use.

Implementation Security and Considerations

Security is not just the mathematical hardness; it also involves ensuring real-world implementations do not introduce vulnerabilities:

- **Randomness:** Kyber requires good random number generation for secrets and noise. A compromised RNG could be disastrous (e.g., if secrets aren't truly random, or if an attacker can predict the noise, they could solve for the secret). This is a general concern for all cryptosystems. Using a NIST-approved DRBG or system entropy is necessary. One might also incorporate some redundancy checks – e.g., some implementations derive all keys from a master seed to ensure they can be regenerated or tested.
- **Error Handling:** In KEM implementations, one must be careful not to leak information through error messages. For instance, if an implementation were to output a special error for a decryption failure (which it should never do except maybe an internal log), an attacker could potentially distinguish that and it would undermine CCA security. Proper practice is to always output a key (valid or pseudorandom) and never indicate to the caller that a ciphertext was invalid.
- **Fault Attacks:** There's a category of attacks where the attacker induces a fault (e.g., via voltage glitching) during computation to see if it reveals the secret. For example, flipping a bit during the NTT or during the error com-

parison might cause a decryption failure or an odd result that reveals something about $\mathbf{s}$. Mitigating this may require redundant computations or error-detection codes in critical steps. It's an active area of research for PQ schemes.

- **Cross-Protocol Attacks:** When integrating Kyber into protocols (like TLS, SSH, etc.), one must ensure proper use. For instance, always authenticate the obtained key (through signatures or MACs) to avoid man-in-the-middle. Use hybrid key exchanges (see Chapter 8) if needed to hedge bets. Also ensure that the chosen parameters match the security level of other primitives in the protocol (no point in a 256-bit quantum-safe key if you then authenticate it with a 128-bit classical cipher without post-quantum security, unless doing hybrid, etc.).

Currently, the **status of Kyber cryptanalysis** is that it is robust and well-understood. Research continues to monitor any advances in lattice algorithms. The community is also exploring post-quantum security in various settings (like how do lattice schemes compose, are there weaknesses in certain lattice structures, etc.), but the consensus is Kyber's Module-LWE problem is solid. NIST's confidence was high enough to push Kyber (ML-KEM) out as a draft standard quickly and finalize it in 2024. Barring a major scientific breakthrough (which would likely affect all lattice schemes), Kyber is expected to provide long-term security against both quantum and classical adversaries.

7

Comparative Analysis of CRYSTALS-Kyber and Other Post-Quantum Algorithms

In this chapter, we compare CRYSTALS-Kyber with several other prominent post-quantum cryptographic algorithms, highlighting their differences in design, performance, and use cases. We focus on **CRYSTALS-Dilithium**, **NTRU**, and **BIKE** as requested, but will also mention other relevant schemes for context.

CRYSTALS-Dilithium (Post-Quantum Digital Signatures)

While Dilithium is a signature scheme (not an encryption/KEM algorithm like Kyber), it is Kyber's sibling in the CRYSTALS suite and also based on lattice cryptography. Dilithium and Kyber share many commonalities: both use module lattices over $\mathbb{Z}_q[x]/(x^n+1)$ and rely on the hardness of the Module-LWE / Module-LWR problems. However, their goals and constructions differ:

- **Purpose:** Dilithium is for **digital signatures**, providing authenticity and integrity (like classical RSA or ECDSA), whereas Kyber is for **encryption/key-establishment**. In practice, one will often use Kyber to establish a session key and Dilithium to sign messages or certificates in a post-quantum secure system.
- **Design:** Dilithium's design is based on the **Fiat–Shamir with aborts** paradigm. It uses a structure related to the **short integer solution (SIS)** problem, another lattice hard problem, which is sort of the "dual" of LWE. In simple terms, a Dilithium public key is a matrix (like in Kyber) and the secret key are short vectors, and a signature is a short lattice vector together with a hint that it's authentic. The scheme involves sampling random challenges and producing proofs that a certain vector is short and related to the secret key, without leaking the key. To keep signatures small and soundness high, Dilithium will sometimes "abort" and retry signature generation if the sample doesn't meet certain criteria (hence the name "Dilithium" implying two-phase like the crystal in Star Trek, as a metaphor).
- **Parameters:** Dilithium uses similar parameters to Kyber: for instance, Dilithium at ~128-bit security (level 2 or 3) uses $n=256$, $q=8380417$ (a larger modulus), and dimension $k=4$ or 5. The modulus is much larger than Kyber's 3329 because signature schemes need to accommodate accumulating more noise as multiple vectors are combined. The larger q also allows a tighter distribution of possible signatures.
- **Performance:** Dilithium is also very efficient. Signing and verifying can be done in under a millisecond on a PC. NIST's final standards included Dilithium as the primary

signature scheme (standardized as **FIPS 204** named "ML-DSA"). Key sizes and signature sizes for Dilithium are on the order of a couple of kilobytes: e.g., Dilithium Level 2 has a ~1312-byte public key and ~2420-byte signature. This is larger than classical signatures (ECDSA sig ~64 bytes) but still quite manageable. Verifying a Dilithium signature involves similar operations (matrix-vector multiplications in a ring) to Kyber encryption, and is very fast (tens of thousands of cycles).

- **Security:** Both Kyber and Dilithium rely on module lattice problems. Dilithium's security is based on hardness of the Module-SIS and Module-LWE problems in rings (it uses a combination in its proof). The cryptanalysis of lattice problems applies to both similarly. A major difference is that signatures remain secure even against a future attacker for already-signed documents (they are not subject to the "record now, break later" issue the way encryption is). So even if lattices were weakened, old signatures remain valid unless the scheme is fully broken. That said, as of now, both Kyber and Dilithium are believed secure with a comfortable margin.

In practice, Kyber and Dilithium complement each other: one provides **confidentiality**, the other **authenticity** in the quantum-resistant realm. They were co-developed and often implemented together, ensuring consistent performance and security levels. It's worth noting NIST recommends Dilithium as the primary quantum-safe signature algorithm, with Falcon and SPHINCS+ as alternatives for specific needs. The use of structured lattices in both Kyber and Dilithium shows the broad applicability of lattice cryptography.

NTRU (Another Lattice-Based Encryption Scheme)

NTRU is one of the oldest post-quantum encryption schemes, first proposed in 1996. It's a lattice-based cryptosystem but with a very different construction than LWE-based schemes like Kyber. Key points about NTRU:

- **Mathematical Basis:** NTRU is built on the problem of finding **short vectors in a different lattice** related to polynomial rings, specifically the convolution polynomial rings. A simple description: NTRU works with polynomials of degree N with small coefficients. A public key is a polynomial $h(x) = f(x)^{-1} * g(x) \pmod{q}$ where f, g are secret polynomials with small coefficients and f is invertible modulo both q and 2. Encryption involves a polynomial combination $h * r + m$ (plus mod operations), and decryption uses the secret polynomial f to recover the message m from that combination. The security is tied to something like finding a short polynomial that is a secret factor of the public key polynomial mod q, which can be seen as a lattice problem (finding (f, g) given h). While not exactly LWE, it's related to the hardness of certain closest vector problems in a cyclic lattice.
- **Evolution:** Over the years, many versions of NTRU emerged. For the NIST PQC process, the submissions included **NTRU-HPS** (named for authors Hoffstein, Pipher, Silverman) and **NTRU-HRSS** (Hoffstein, et al.), as well as a variant called **NTRU Prime** that aimed to avoid structures that could be attacked (using a different ring $Z_q[x]/(x^n - x^{n/2} - 1)$ to break symmetries). NTRU was a finalist alongside Kyber and Saber. NIST ended up not choosing

NTRU in the first batch (preferring Kyber), but **NTRU is essentially considered a backup candidate** – NIST stated that if something unforeseen had arisen with Kyber (like a patent issue or a major break), they might standardize NTRU instead. In fact, NTRU's long history and public domain status (the core patents expired in 2017) were factors in its favor.

- **Performance and Sizes:** NTRU encryption is fast but historically a bit slower than Kyber. As noted, NIST found its keygen slower and its keys/ciphertexts somewhat larger. Concretely, one NTRU parameter set (ntruhps2048–677, targeting ~128-bit security) has a public key of 935 bytes and ciphertext of 699 bytes, which are actually *slightly smaller* than Kyber512's 800/768 bytes. Another variant (ntruhrss701) has pk of 1138 bytes, ct 1138 bytes, closer to Kyber768 sizes. So size-wise NTRU is in the same ballpark, if not a bit better for Level 1. But NTRU's computation involves large-degree polynomial multiplications (e.g., degree 701) using convolution, which can be efficiently done with FFTs too. When optimized, NTRU is quite competitive; it was maybe a factor of 1.2–1.5× Kyber's runtime in many benchmarks. Importantly, as message sizes are small (a few hundred bytes at most), the differences in speed are relatively minor for most applications (both schemes complete handshakes in milliseconds or less).
- **Security Assumption Differences:** A reason NIST gave Kyber a slight edge is trust in its underlying assumption. Kyber relies on Module-LWE, which has a strong theoretical lineage to worst-case lattice problems with proof reductions. NTRU's hardness is more *heuristic* – it's not proven reducible to worst-case lattice problems in the

same way. However, NTRU has stood the test of time (no big structural breaks in over 25 years, which is a kind of empirical evidence). NIST found MLWE "marginally more convincing" than NTRU's assumption, but not by a huge margin. Both are believed secure against known quantum and classical attacks at appropriate parameters.

- **Usage:** If Kyber is standardized as expected (FIPS 203) and deployed widely, NTRU might see less use. However, some organizations may choose to implement multiple KEMs for agility. NTRU could be a fallback in case any weakness is ever found in Kyber. Also, being older, NTRU might appear in certain protocols or as a check-box for compliance. It's worth noting that NTRU was also submitted to international standards bodies (like ANSI and IEEE in the past, and now likely through ISO channels). In any case, from a performance and integration perspective, using NTRU vs Kyber is not drastically different – they both fit similar roles. NTRU's biggest downside historically was patent encumbrance, but that's no longer an issue; both Kyber and NTRU are unencumbered (Kyber's implementing algorithm is free to use and NTRU's patents expired or were released).

BIKE (Bit-Flipping Key Encapsulation) – Code-Based KEM

BIKE is a code-based cryptosystem, meaning its security comes from error-correcting code problems (specifically, a variant of the NP-hard syndrome decoding problem on linear codes). It was an alternate finalist in the NIST competition and remains under consideration in the fourth round for potential standardization as a backup **non-lattice KEM**.

Key features of BIKE:

- **QC-MDPC Codes:** BIKE uses **quasi-cyclic moderate-density parity-check** codes. Without diving into coding theory jargon, BIKE's public key can be seen as a parity-check matrix of a linear code that has a certain structure (quasi-cyclic means it's composed of circulant blocks). The private key is basically two cyclic binary polynomials (or bit strings) that serve as the secret parity-check matrix. Encryption in BIKE is analogous to the Niederreiter encryption: a message is encrypted by multiplying it with the public parity-check matrix and adding random error of a fixed weight t. Decryption uses a bit-flipping decoder (an iterative algorithm) with the private key to recover the error vector and thus the message.
- **Performance:** BIKE has an interesting performance profile. Its **encryption and decryption** are essentially just binary vector operations (XORs, shifts) and a decoding loop that is quite fast on average. In software, BIKE can encrypt/decrypt quickly – often comparable to lattice schemes. In fact, the NIST report noted *"BIKE is one of the more efficient alternate KEM candidates"* and that *"hardware benchmarks confirm performance of BIKE would be suitable for most applications"*. One metric given was that except for SIKE, data transmission costs dominate CPU cost for all candidates including BIKE, meaning BIKE is not extremely slow. However, BIKE's **key generation** is relatively slow because it involves finding a pair of polynomials with certain weight properties that are also inverses mod x^r-1. This can take many tries. NIST noted BIKE's keygen *"runs significantly slower than other structured code- and*

lattice-based schemes". Fortunately, keygen is usually done infrequently.

- **Sizes:** BIKE's sizes are a bit larger than Kyber's but not exorbitant. BIKE Level-1 (128-bit sec) has public key and ciphertext on the order of a few thousand bits (e.g., around 1300–1400 bytes for pk, similar for ct). This is perhaps 1.5×–2× Kyber's sizes, but still easily within single packet size. HQC, another code-based alternate, has even larger sizes (pk ~3500 bytes, ct ~7000 bytes at level 1), so BIKE is more bandwidth-efficient than HQC. Compared to NTRU/Kyber, BIKE's overhead is somewhat higher in bytes, but not an order of magnitude. For instance, one BIKE param (BIKE-1 Level 1) had pk ~1042 bytes, ct ~1834 bytes, which is more than Kyber512's 800/768 but still reasonable.
- **Security:** BIKE's security is based on the difficulty of decoding random linear codes with moderate redundancy – specifically, decoding a random quasi-cyclic code with moderate weight noise. This relates to the well-studied **syndrome decoding problem**, which is believed to be hard even for quantum computers (quantum gives only a slight speedup via Grover search in generic decoding). The BIKE team had to ensure that the probability of decryption failure (i.e., decoding failure) is negligible, as a high failure rate could be turned into a powerful attack. There were investigations into BIKE's decoding failure rate and potential vulnerabilities. By tweaking the algorithm (e.g., randomizing certain decoding steps) and parameter choices, BIKE now claims a negligible failure probability and IND-CCA2 security. Code-based schemes have a long history (e.g., McEliece in 1978) and no one has

found a way to break them significantly faster than brute force or information-set decoding algorithms. So BIKE's assumptions are considered very different from lattice ones, providing a hedge in case some unforeseen quantum algorithm attacks lattices.

- **Use cases:** If standardizing a second KEM, NIST might choose BIKE or HQC in 2024 for diversity. BIKE is attractive for its simplicity and relatively small footprint among code-based options. However, one must accept a slower keygen (for ephemeral keys this might be fine) and larger ciphertext. BIKE might be preferred in scenarios where one fears that lattice schemes could be broken (though that's deemed unlikely currently), or where patents/IP on lattice schemes are a concern (currently Kyber/NTRU are not patented, but historically some lattice schemes were). BIKE is free of such issues.

In comparison to Kyber: Kyber has the edge in **key size, speed, and familiarity** (more cryptographers are versed in lattice vs coding theory nowadays). BIKE has the edge in providing a very different hardness assumption and slightly simpler arithmetic (binary). Both appear to be solid; Kyber is "main event" and BIKE is the insurance policy, to paraphrase NIST's statements.

Other Noteworthy Post-Quantum Schemes (Briefly)

While the question highlights Dilithium, NTRU, and BIKE, for completeness it's worth mentioning:

- **Falcon (Signature):** Another lattice signature, using Fourier sampling, with smaller signatures (~666 bytes) but

requiring floating-point arithmetic for signing. Chosen by NIST as an alternate signature standard (FIPS 205 in future) for applications needing smaller signatures than Dilithium.

- **SPHINCS+ (Signature):** A stateless hash-based signature scheme. Very slow and large signatures (~10k bytes), but based only on hash functions (very conservative security). Standardized as well by NIST as an alternate, mainly for its different assumption (no reliance on lattices or any new math)
- **Classic McEliece (KEM):** Code-based encryption using binary Goppa codes. Has gigantic public keys (at level 5, nearly a megabyte) but tiny ciphertexts (128 bytes) and very high security confidence from decades of scrutiny. Still under consideration as an alternate; niche use due to key size.
- **HQC (KEM):** Another code-based KEM like BIKE, with different code construction (uses LDPC codes). Has larger bandwidth but simpler constant-time decoding. Competing with BIKE in the alternate pool.
- **FrodoKEM (KEM):** A lattice (plain LWE) KEM that doesn't use rings ("unstructured" lattices). Highly secure assumptions but very slow and big (public key ~10KB for 128-bit sec). NIST gave it alternate status in Round 2 but it's on hold now. It showed the difference structure makes: Kyber (structured) vs Frodo (unstructured) was night-and-day in performance.

Each scheme has trade-offs, but the general trend is that **lattice-based cryptography (Kyber, Dilithium, NTRU, Saber, Falcon)** tends to offer the best mix of efficiency and acceptable

key sizes, which is why lattice schemes dominate the first-wave standards. Code-based schemes (BIKE, HQC, McEliece) serve as important alternatives for safety. Hash-based and multivariate schemes play smaller roles.

To conclude the comparative analysis: CRYSTALS-Kyber stands out as the *de facto* standard for post-quantum encryption (key exchange) due to its strong security basis and excellent performance. CRYSTALS-Dilithium similarly is emerging as the standard for post-quantum digital signatures. NTRU remains a viable lattice KEM with a long pedigree, slightly less optimal but still strong, and exists as a backup. BIKE offers a compelling non-lattice approach, competitive in performance and likely to be standardized for diversification. Together, these algorithms represent the new toolkit of cryptography for the quantum age.

8

Adoption and Standardization Status

Post-quantum cryptography is transitioning from research to real-world deployment. In this chapter, we overview the adoption status of CRYSTALS-Kyber and related PQC algorithms, focusing on standardization processes and regulatory developments globally. We cover NIST's PQC standardization in the US, as well as initiatives in the EU, China, and other regions, reflecting the worldwide push towards quantum-resistant security.

NIST Standardization and Recommendations

The United States, through NIST, has been leading the PQC standardization effort. The process began with a call for algorithms in 2016, and after three rounds of evaluation, NIST announced the first selections in July 2022. CRYSTALS-Kyber was selected as the **primary algorithm for public-key encryption/KEM** (Key Encapsulation Mechanism) to be standardized. Alongside Kyber, NIST chose three signature algorithms (CRYSTALS-Dilithium, FALCON, and SPHINCS+).

NIST has since worked on turning these into official FIPS (Federal Information Processing Standards). In 2023, NIST released draft standards for Kyber (encryption) and Dilithium/SPHINCS+ (signatures). By August 2024, NIST **finalized the standards**, assigning Kyber the designation **FIPS 203, "Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)"**. (NIST gave Kyber a more generic name in the standard to emphasize the class of scheme and version, but it is essentially Kyber.) Dilithium became FIPS 204 (ML-DSA) and SPHINCS+ FIPS 205. These standards include the algorithm specifications, implementation guidelines, test vectors, etc., and are now ready for use by government agencies and industry.

NIST's messaging has been clear: **"Don't wait – start implementing post-quantum algorithms now."** NIST and other U.S. agencies like NSA and CISA have urged organizations to begin migration well ahead of the arrival of quantum computers. They warn of the "store now, decrypt later" threat: adversaries can intercept and save encrypted data today, with the intent to decrypt it in the future when quantum capabilities might be available. Thus, data that needs long-term confidentiality (sensitive government data, personal health data, intellectual property, etc.) should be protected with quantum-resistant algorithms as soon as practical.

To guide the transition, NIST's National Cybersecurity Center of Excellence (NCCoE) has published migration frameworks and best practices. The NSA, CISA, and NIST released a joint advisory in August 2023 with recommended steps for organizations to achieve "quantum readiness":

- **Inventory and Crypto Audit:** Identify where and how cryptography is used in systems (protocols, applications,

hardware).

- **Develop a Roadmap:** Make a plan for replacing vulnerable algorithms with PQC, prioritizing systems that protect long-lived secrets.
- **Engage Vendors:** Work with technology providers to ensure they have solutions (e.g., a TLS library supporting Kyber, or HSMs with PQC).
- **Test and Implement:** Start integrating PQC algorithms in test environments, ensure interoperability and performance, then deploy gradually.
- **Stay Agile:** Adopt "crypto-agility" – systems should be designed to swap out cryptographic components with minimal disruption. That way, if any issue arises with a chosen PQC algorithm or a new one emerges, the update is manageable.

The U.S. government itself is following a directive from the White House. A National Security Memorandum (NSM-10, May 2022) set timelines for federal agencies to inventory their cryptographic systems and have plans to transition to PQC by 2025, and to actually move to compliant systems by 2035 at the latest. Some critical systems (especially in intelligence/military) likely will transition much sooner. NSA's Commercial National Security Algorithm Suite 2.0 (CNSA 2.0) has already announced inclusion of algorithms like Kyber and Dilithium for national security systems.

To facilitate industry adoption, NIST has been proactive in outreach. They've published documents like NIST IR 8413 (third-round report) summarizing rationale and urging adoption. They emphasize that these new standards are ready for integration and that waiting could leave a security gap.

International and Industry Adoption

ISO/IEC – The international standardization bodies are also working on PQC. Typically, ISO will produce standards that are often aligned with NIST's outcomes (since many cryptographers overlap). Work is underway in ISO/IEC JTC 1/SC 27 to standardize lattice-based schemes including Kyber (often under a different name for IP reasons, but essentially the same algorithm). These ISO standards will help countries that prefer ISO endorsement to adopt PQC.

IETF – The Internet Engineering Task Force is actively drafting standards to use PQC in Internet protocols. For example, IETF has drafts for using hybrid ECDH+Kyber in TLS 1.3, for using PQC in IPSec/IKEv2, and for encoding PQC public keys and certificates. A specific draft (draft-uni-qsckeys-kyber) addresses how to identify and serialize Kyber keys in various formats to ensure interoperability. The IETF has already published RFC 9180 (Hybrid KEM for HPKE) which includes Kyber variants combined with X25519, for use in secure messaging protocols. We can expect within a year or two official RFCs that add PQC cipher suites for TLS, and an update to X.509 certificate standards to allow PQC public keys and signatures (some of these are already being tested; e.g., there's a draft RFC for encoding Dilithium/Falcon in X.509).

Industry Pilots – Big tech companies and cryptographic library maintainers have been testing PQC algorithms in practice. For instance, Cloudflare and Google conducted experiments in 2019 and 2022 where they added a post-quantum key exchange (like CECPQ2 which used Kyber) to TLS connections to gauge performance and compatibility. Many of these trials went smoothly, demonstrating that algorithms

like Kyber can be integrated with minimal issues. Libraries like OpenSSL, BoringSSL, and mbedTLS have beta support for Kyber and Dilithium (often under compile-time options, as formal standards were pending). The Open Quantum Safe (OQS) project provides a library (liboqs) and OpenSSL fork that implement Kyber, Dilithium, etc., making it easier for developers to prototype with PQC. These efforts mean that once standards are finalized, production-ready implementations will quickly follow.

Hardware manufacturers are also getting involved. For example, some smart card and secure element makers (Infineon, NXP, etc.) have announced support for PQC in their products. We even see FPGA IP cores and prototype ASIC designs for accelerating Kyber. In 2025-2026, we might see the first TPMs (Trusted Platform Modules) and HSMs (Hardware Security Modules) that offer Kyber-based key generation and key agreement, which is crucial for integration into secure hardware systems.

Certification bodies (like Common Criteria, FIPS 140-3) are also updating their guidelines to include PQC algorithms. FIPS 140-3 testing will need to include testing the correct implementation of FIPS 203 (Kyber). Common Criteria modules for cryptographic primitives are being revised to incorporate PQC, ensuring that products that go through CC evaluation can claim compliance with PQC requirements.

European Union Initiatives

The EU has been very proactive in post-quantum cryptography transition. The European Commission and ENISA (European Union Agency for Cybersecurity) have issued strategy

documents and recommendations for member states to move to quantum-safe crypto. In 2022, the EU launched a "Joint Cyber Unit" initiative and a "**Coordinated PQC Transition**" **roadmap**. According to an EU Digital Strategy document, *"All Member States should start transitioning to post-quantum cryptography by the end of 2026"*. Moreover, critical infrastructures and government services are urged to be quantum-resistant by 2030.

The EU's roadmap (sometimes called the EU PQC Toolbox) includes:

- Awareness campaigns and skill-building for organizations about PQC.
- Identifying critical sectors (finance, health, energy) that need early adoption.
- Possibly mandating PQC compliance in procurement for government systems beyond a certain date.
- Considering interoperability across the union – ensuring that, say, a digital signature from one country's system (using Dilithium) is recognized by another's.

ENISA has published a "**Recommendations on Quantum-resistant Cryptography**" which align largely with NIST's selections. They also stress investigating European contributions and keeping an eye on alternate algorithms (Europe had significant involvement in algorithms like Classic McEliece and HQC too). The Horizon Europe research funding has multiple projects (like PQC4EU, PROMETHEUS) that did R&D and proof-of-concepts for PQC in European contexts.

There's also an interesting interplay with **Quantum Key Distribution (QKD)** in Europe. The EU often mentions

QKD (physical-layer quantum encryption) as a complementary technology. However, QKD has very different use cases and requirements; PQC is the primary solution for most digital communications. The EU is funding both, but PQC is far easier to deploy broadly.

Developments in China, Russia, and Others

China: China has been active in cryptography standardization through its organization, the Chinese Association for Cryptographic Research (CACR). In 2022, reports emerged that China is launching its own post-quantum standardization project, somewhat independent of NIST. The **Institute of Commercial Cryptography Standards (ICCS)** in China issued a call for proposals for quantum-resistant algorithms in early 2025. This move might be driven by a desire for technological sovereignty (not relying on "Western" standards) and concerns (voiced by Chinese experts) that NIST's process might have hidden weaknesses or backdoors.

China has not yet announced final picks, but they have domestic algorithms in consideration. For example, Chinese researchers have worked on lattice algorithms like "Rainbow" (for signatures, though that was broken) and others, as well as isogeny and multivariate schemes. As of late 2025, it's reported that China had not publicly selected its PQC standards yet. However, historically, China often ends up standardizing algorithms similar to international ones, perhaps with minor tweaks or in parallel. Dustin Moody of NIST noted that China previously selected algorithms analogous to NIST's but in a more closed way. So it's plausible China might adopt something like Kyber or a variant thereof under a different name, unless

they have a homegrown alternative they trust more. The development bears monitoring because if China diverges (say they pick completely different lattice parameters or a different system), it could complicate global interoperability.

Chinese regulations also likely will mandate quantum-safe crypto for government and critical infrastructure by some timeline (potentially even earlier than US/EU in some cases, as they can enforce changes top-down). There's mention that China is promoting both PQC and QKD for high-security scenarios.

Russia: Russia historically has its own set of GOST algorithms. It's not clear if they have a specific PQC standard effort publicly known. Given the international situation, they might not participate in NIST/ISO, but Russian academia is involved in PQC research (there were some Russian teams in NIST competition too). Likely, Russia will eventually adopt some lattice-based schemes (perhaps even Kyber/Dilithium under different names) through their standards body (FSTEC/FSB). They might also lean on domestic mathematical research – but since lattice cryptography is largely public domain, they may just implement what the world does, possibly with modifications to align with any national policy.

Japan, Korea, others:

- *Japan* has been aligned with NIST process; many Japanese researchers contributed to algorithms. Japan's CRYPTREC advisory list will update to include PQC. Japan is also home to many device manufacturers (Renesas, Toshiba) who are implementing PQC.
- *South Korea* held a competition in parallel to NIST called "PQC Competition Korea" and reportedly selected two al-

gorithms: **HAEDEL** (which is essentially Kyber with a twist, I believe) and **LEA** (actually LEA is a block cipher; need to confirm: the search snippet suggests "Korea has selected HAETAE and …" possibly for signature?). Possibly they chose a variant named "HAETAE" (the Korean mythical creature) as a KEM. These might be based on the same principles as Kyber or others, or could be independent. But Korea will also likely accept NIST algorithms for international compatibility (e.g., for TLS).

- *Australia, Canada, UK*: These typically follow NIST guidance for cryptographic standards (especially since they partner on Five Eyes signals intelligence, and NSA has endorsed NIST's choices). Indeed, the UK's National Cyber Security Centre and Canadian CSE have published guidance mirroring NSA's call to migrate to NIST PQC.

Patent and IP Issues: One notable aspect of adoption is ensuring algorithms are patent-free or licenses are clear. Kyber's components (like NTT, etc.) are all old math, and the reference implementation is public domain. The Kyber team made a patent declaration that there are no known encumbrances. NTRU had old patents but they expired or were released. BIKE/HQC have no known patents. This smooths adoption. NIST required submitters to either have patents available royalty-free or none at all. So standards adoption shouldn't be held back by IP—unlike say the AES process where Rijndael had no patents, PQC had to navigate some patent minefields (like one alternate candidate had issues and was dropped partly due to that, e.g., some supersingular isogeny components had patents). But Kyber is clear. NIST did condition Kyber's standardization on finalizing a few IP

agreements – likely ensuring any contributors with possible claims formally relinquished them.

Overall Readiness and Global Regulations

The **global regulatory environment** is moving toward mandating quantum-safe encryption in various sectors:

- In the **financial sector**, regulators are asking banks and payment systems to assess quantum risk and have a migration plan (for instance, US FDIC and Fed have mentioned this in risk outlooks; European Central Bank too).
- In **healthcare** and **data protection** (e.g., GDPR in Europe), while not explicitly mentioning PQC yet, the general requirement to use "state of the art" security could be interpreted to include PQC once standards are mature, especially for data that must remain confidential for many years (like genomics data, which could still be sensitive 20 years from now).
- **Critical infrastructure** regulations (energy grids, telecom) in various countries are being updated to include cryptographic agility and PQC preparedness.
- The **US government** has binding operational directives that will likely require federal agencies to use FIPS 203 (Kyber) for certain types of new systems by a set date once FIPS 203 is fully approved (likely 2024/2025).
- **EU law** may incorporate PQC in updates to eIDAS (electronic identification and trust services) for digital signatures; eIDAS already references specific algorithms for qualified signatures, and an update could add Dilithium/Falcon as allowed algorithms for qualified electronic signatures in

the EU.

All these moves underscore a consensus: **the time to migrate to post-quantum cryptography is now**. There is high-level political awareness too – for example, the U.S. National Security Strategy and EU's Digital Compass both mention quantum-safe cryptography as critical to future security. Government funding has been allocated not just to research but to help with the transition (like grants for upgrading infrastructure, competitions to encourage PQC solutions, etc.).

From an adoption perspective, it appears that by 2025:

- Standards (NIST FIPS, IETF RFCs, ISO standards) for PQC are largely in place or imminent.
- Companies are beginning to include PQC in their products (some VPNs advertise PQC support, cloud providers like AWS and Google offer options for PQC key exchange in certain services).
- Regulations are in the process of being updated to require these algorithms over the second half of the 2020s.

It's a complex coordination effort, but one can draw parallels to the migration from 1024-bit RSA to 2048-bit RSA, or from SHA-1 to SHA-256 in the past – except this is a bigger shift because the algorithms are entirely different. Still, the momentum is there. As NIST's Dustin Moody said, *"these new standards are the main event... Go ahead and start using these three [Kyber, Dilithium, SPHINCS+]"*, with more algorithms to come as backups. The world is heeding that advice, moving toward a quantum-resistant security foundation.

9

Performance Benchmarks of Kyber Across Platforms

Kyber is widely liked in real systems because it is fast, memory-friendly, and has relatively small keys and ciphertexts compared to many post-quantum alternatives. In practice, most of Kyber's cost comes from two places: polynomial arithmetic (especially NTT-based multiplication) and hashing (SHAKE functions). When engineers optimize Kyber, they usually optimize those two components first.

Desktop CPU performance (Intel and AMD, x86-64)

On modern desktop and server CPUs, highly optimized Kyber512 implementations can complete each operation in tens of thousands of CPU cycles.

Typical optimized cycle counts reported for Kyber512 on an Intel Haswell class CPU with AVX2 are:

- Key generation: about 34,000 cycles

- Encapsulation: about 45,000 cycles
- Decapsulation: about 35,000 cycles

Larger variants do more polynomial work because the parameter k is larger, so they cost more cycles. A commonly cited number for Kyber1024 encapsulation is about 97,000 cycles when optimized.

This is a big deal because it means the cryptographic delay is usually in the microseconds range on a modern core. In protocols like TLS handshakes, network latency is often the dominant delay, not the Kyber computation itself.

Why vector instructions matter (SIMD and AVX2)

Kyber's arithmetic involves many small independent operations on integers, which is perfect for SIMD vectorization. AVX2 lets the CPU process many 16-bit values in parallel, speeding up the NTT and related routines.

This is why optimized code can be dramatically faster than "plain C" code. A representative comparison for Kyber512 decapsulation is:

- Plain C: about 187,000 cycles
- AVX2 optimized: about 34,000 cycles

That is more than a 5 times improvement, mostly from better NTT kernels and better data movement.

Bandwidth and object sizes (public keys, secret keys, ciphertexts)

Kyber's sizes are small by post-quantum standards, which is one reason it integrates nicely into network protocols.

Common size figures:

- Kyber512: public key about 800 bytes, secret key about 1,632 bytes, ciphertext about 768 bytes
- Kyber768: public key about 1,184 bytes, secret key about 2,400 bytes, ciphertext about 1,088 bytes
- Kyber1024: public key about 1,568 bytes, secret key about 3,168 bytes, ciphertext about 1,568 bytes

A practical interpretation: the public key plus ciphertext is only a few kilobytes total, which typically fits comfortably within normal network packet behavior. That means you usually pay more in round trips and latency than in raw transmission time.

Throughput vs latency (what servers care about)

Because each operation is fast, a single server core can potentially handle very large numbers of Kyber handshakes per second under ideal conditions. A commonly stated ballpark is on the order of 100,000 Kyber handshakes per second per core, not counting network waiting and other application overhead.

Embedded and mobile performance (Cortex-M class devices)

Kyber has also been demonstrated on constrained devices like ARM Cortex-M4.

Representative figures reported for Kyber512 on a 24 MHz Cortex-M4:

- Encapsulation: about 56,000 cycles, which is about 0.00233 seconds, or about 2.33 milliseconds
- Decapsulation: about 530,000 cycles, which is about 0.02208 seconds, or about 22.08 milliseconds

These numbers imply that many IoT devices running faster clocks, like 48 MHz to 120 MHz, can often complete a key agreement in a few milliseconds, which is practical for firmware updates, sensor networks, and device-to-device secure channels.

Memory footprint (RAM and flash)

Memory is often a bigger constraint than raw speed on microcontrollers. Optimized implementations can reduce RAM dramatically by reusing buffers and using in-place transforms.

Example figures often cited for optimized Kyber512 on Cortex-M4:

- Working RAM during operation: about 2.5 kilobytes
- Flash footprint (code plus tables): often around 20 to 30 kilobytes

Unoptimized reference implementations can use significantly

more RAM, sometimes in the range of 8 to 14 kilobytes depending on the operation, which can be too large for very small chips. Optimization focuses heavily on reducing temporary buffers.

Energy cost (mobile reality)

Energy depends on hardware, but a commonly reported ballpark comparison in mobile contexts is that using Kyber in place of classical elliptic-curve key exchange can increase CPU energy cost by roughly 2.3 times for the key exchange portion. In many real applications, the overall battery impact can still be modest because networking and symmetric encryption often dominate.

Hardware acceleration (FPGA and ASIC friendliness)

Kyber's structure is regular and hardware-friendly: lots of predictable modular arithmetic, lots of repeated kernels, and clear opportunities for parallelism. FPGA implementations can reach thousands of operations per second with relatively small area and power, making PQC coprocessors plausible for routers, smart cards, and embedded security modules.

Comparison with other post-quantum KEMs (engineering view)

- Compared to NTRU: Kyber is often faster in key generation and has very competitive overall handshake cost.
- Compared to Saber: performance is frequently close, with the winner depending on architecture and implementation details.
- Compared to code-based schemes like BIKE and HQC: those can be fast in some operations but typically have larger keys and ciphertexts, and some have very slow key generation.
- Compared to SIKE: SIKE was far slower and was broken in 2022, so it is no longer relevant as a secure option.

Bottom line

Kyber's performance profile is one of the reasons it became the default "workhorse" post-quantum key exchange candidate: low latency on desktops, workable timing on microcontrollers, small bandwidth, and a structure that rewards both software and hardware optimization. In many deployments, the real bottleneck is not Kyber itself, but network delays, I/O, and system-level overhead.

10

Practical Deployment Scenarios and Migration Strategies

Transitioning to post-quantum cryptography involves not only choosing secure algorithms like CRYSTALS-Kyber, but also integrating them into complex real-world environments. In this chapter, we discuss how PQC (with a focus on Kyber) can be deployed in practice, the interoperability challenges that may arise, and strategies for migrating from classical to post-quantum cryptography.

Deployment Scenarios for CRYSTALS-Kyber

Kyber, as a KEM, is mainly deployed to protect **data in transit**, establishing shared keys for secure communication. Some key scenarios include:

- **Secure Web Traffic (TLS):** Kyber can be used in the TLS handshake to negotiate a session key. The likely deployment mode is a hybrid key exchange: e.g., the client and server exchange both an ECDH (Curve25519 or P-

256) and a Kyber encapsulation, and derive the final master secret from both. This hedges against any unforeseen weakness in either algorithm. Major browsers and servers have tested such hybrids successfully (e.g., Chrome experiments with CECPQ2 combining X25519+Kyber512). Once confidence is extremely high in PQC, pure Kyber key exchange could be used on its own, simplifying the handshake. In TLS 1.3, a Kyber KEM would fit in the "key_share" extension (just like an X25519 share today, except it's not a share but a KEM; in practice, the client would send a Kyber public key, and server responds with a Kyber ciphertext encapsulating the secret). Draft proposals define exactly how to encode and handle this.

- **VPN and Secure Tunnels:** Protocols like IKEv2/IPsec for VPNs or SSH for remote login can similarly use Kyber for key exchange. For instance, SSH could adopt a new key exchange method "sntrup761x25519-sha512@openssh.com" which combines NTRU or Kyber with X25519 (OpenSSH has already added an experimental NTRU+X25519 option). In IPsec's IKE, proposals for PQC are being standardized so that two gateways can establish a quantum-safe session key for IPsec tunnels, protecting corporate site-to-site communications from future quantum decryption attempts.
- **Secure Email (S/MIME, PGP):** Email encryption could use Kyber to encrypt the symmetric content key that protects the email body. PGP, for example, could generate a random session key, then use the recipient's Kyber public key to encapsulate that session key, including the resulting ciphertext in the message. S/MIME (X.509 certificates) could do similar – we might see X.509 certificates that contain a Kyber public key (for key agreement) and perhaps

a Dilithium public key (for signature) for a user. This would allow end-to-end quantum-safe email encryption.

- **IoT Communications:** Resource-constrained IoT devices (sensors, smart home gadgets) that use protocols like MQTT or COAP could employ Kyber for establishing keys with servers. Many IoT devices need long-term security (for example, a smart meter's firmware updates should remain secure for 10-20 years). Kyber's efficiency on microcontrollers makes it feasible for IoT. One could imagine a future standard of Bluetooth or ZigBee that includes a PQC key exchange for pairing devices.
- **Secure Storage and Key Agreements for Data at Rest:** While Kyber is not directly used to encrypt bulk data (you'd typically use AES for that), it can be used to protect keys. For instance, a cloud storage service might protect a user's file encryption keys by wrapping them with a Kyber KEM to the user's public key, ensuring that only the intended user (with the private key) can recover the file key – and not even a future quantum adversary who steals all stored data can decrypt it without also breaking Kyber.
- **PKI and Certificates:** Deploying Kyber in practice requires infrastructure support. Certificate Authorities might begin issuing **hybrid certificates** which contain both a classical public key (for use by systems that don't support PQC yet) and a PQC public key. Alternatively, two separate certs might be used. There's an ongoing debate on how to transition PKI – whether to go all-in with PQC keys or run dual infrastructures. Some early adopters have used composite certificates that essentially glue an RSA and a PQC signature together so they can be verified by either method. The same can apply for encryption: a certificate

could include an RSA key (2048-bit) and a Kyber key; a sender could choose which to use based on capability. Long term, of course, the classical key would be dropped.

- **Hardware Security Modules (HSMs) and Smart Cards:** Deployment in environments like banking or identity may require hardware support. We expect new HSM models that support Kyber key generation, encapsulation, decapsulation inside a secure module. Similarly, smart cards or secure elements (like those in passports or TPM chips in computers) will need to implement Kyber if they are to do key establishment. Some secure element vendors have already demoed PQC on chips. One challenge here is that PQC algorithms, being somewhat heavier, might be slower on small chips, but as discussed, Kyber is quite efficient and likely within the capability of even a modest smart card CPU (though maybe not as fast as RSA-2048 with CRT optimizations on a crypto chip, which is extremely optimized; but adequate).
- **Government and Military Communication:** These entities will be among the first to fully deploy PQC for classified or sensitive comms. Expect Kyber (ML-KEM) to be incorporated into secure radio systems, satellite communications, etc. The US DoD, for instance, might mandate that all its new cryptographic devices be NSA Suite 2.0 compliant, which includes Kyber. NATO and other alliances will coordinate on algorithm profiles to ensure interoperability between nations' secure equipment.

Interoperability Challenges

Migrating to new cryptography at Internet-scale introduces numerous challenges:

- **Protocol Compatibility:** Many protocols have size constraints and assumptions baked in. For example, some might assume a handshake message can fit in a certain number of bytes. PQC keys and ciphertexts are larger (though Kyber is still small enough for most). But consider, e.g., if a protocol used fixed-size fields for public keys of 256 bytes and Kyber's is 800 bytes – that protocol needs extension. Thus, protocols must be revised (TLS 1.3, IPsec, etc. are fortunately extensible by design to accommodate larger keys). We must also ensure negotiation mechanisms can handle new algorithm identifiers.
- **Mixed Environments:** During the transition, not everyone upgrades at once. There will be clients/servers that support PQC and those that do not. This raises compatibility questions: for instance, if a web browser supports Kyber but the server does not, they'll fall back to a classical ECDH – which is fine, but then that connection isn't quantum-safe. If the client absolutely requires quantum safety (maybe a setting for very sensitive data), it might refuse connections without PQC, but then it can't communicate with legacy servers. The pragmatic approach is for some years to come, systems will run in hybrid modes or allow classical fallback, accepting the interim risk for interoperability. Gradually, as virtually all major products support PQC, one can start disabling the old algorithms.
- **Performance Tuning:** Introducing PQC can affect per-

formance. We noted earlier that using Kyber in TLS might double the handshake CPU usage and significantly increase the handshake message size (though still quite small in absolute terms). On high-latency links, bigger handshake messages might slightly increase handshake time (70x more data overhead, but if that's only a few KB vs a few dozen bytes, on a 100Mbps link it's negligible, on a low-bandwidth or high-latency link it could add some hundred milliseconds). There might be corner cases where, say, thousands of IoT devices all initiate Kyber handshakes at once and a server's CPU gets more loaded than with ECDH. Careful capacity planning and possibly hardware acceleration (e.g., offloading PQC operations to vector units or cryptographic co-processors) might be needed for heavy use cases like VPN concentrators or large web servers. In testing, however, it appears well-optimized PQC can handle typical loads. Cloudflare noted only minor impact on their server performance when enabling hybrid X25519+Kyber vs X25519 alone.

- **Certifications and Compliance:** Some industries require certified crypto implementations (FIPS 140, Common Criteria, etc.). Initially, PQC implementations might not be certified, which could slow adoption in regulated sectors. However, NIST is updating FIPS 140 validation to include PQC algorithms, and labs are tooling up to test them. Another angle: some devices (like payment systems) have hard-coded algorithm lists allowed (e.g., EMV for credit cards). Updating those standards to permit PQC will take time, and until then, including PQC might violate compliance. Industry consortiums (like PCI for payments, IEEE for certain networks, etc.) will need to update their

standards.

- **Key Management and Infrastructure:** Organizations must adapt their key management. For instance, if you start issuing Kyber public keys, how do you distribute and store them? HSMs or databases might need schema changes to accommodate different key sizes and types. Backup and recovery processes for keys need to consider PQC keys (though they're similar to others, just bigger). Also, how to handle existing data that was encrypted with RSA or EC? It might need re-encryption with PQC if it needs to remain secure long-term. This kind of large-scale re-encryption (think: millions of stored files or database entries) can be very time-consuming. A phased approach might encrypt new data with PQC keys while leaving old data as is if it's expected to be pruned before quantum threats are real. Alternatively, truly long-lived data should be proactively re-encrypted now with PQC algorithms or at least protected via a hybrid scheme (encrypt with AES under an RSA+Kyber hybrid envelope now, so even if RSA breaks, the Kyber part still protects the symmetric key).
- **Human factors and Training:** Another interoperability aspect is ensuring that IT personnel and developers are familiar with these new algorithms. The terminology, error rates, etc., are new. Mistakes in implementation could cause outages (imagine a sysadmin deploying a TLS cert with a Kyber key to a server that doesn't yet support it – the site might become unreachable to users who can't handle it). Gradual rollout in test networks, and clear guidelines, are important.

Migration Strategies and Best Practices

Migrating to PQC is a significant project, and experts have outlined strategies to manage it effectively:

- **Prioritize "Store Now, Decrypt Later" Risks:** As mentioned, data that, if compromised in 10-15 years, would be catastrophic, should be prioritized. This includes things like government secrets, health records, personal data protected by regulations (which often require confidentiality for decades). Those communications should move to hybrid or PQC as soon as possible. Conversely, some short-lived secrets (like a session cookie that expires in an hour) might not need immediate PQC, since even if an adversary records it, by the time they might decrypt it with a quantum computer, it's useless. However, since you usually upgrade systems wholly rather than by data type, this distinction mostly guides which projects to focus on first.
- **Use Hybrid Encryption During Transition:** For a period (likely this entire decade), using hybrid solutions (classical + PQC) provides defense-in-depth. For example, TLS can derive keys from both an ECDH and a Kyber exchange. An attacker would need to break both. This ensures security even if one algorithm fails. The downside is complexity and some performance cost, but it's worth it for critical systems initially. Hybrid mode can be phased out later when confidence in PQC is universal.
- **Maintain Crypto-Agility:** Perhaps the biggest lesson is to design systems that are not hardcoded to specific algorithms. Many organizations got burned by hardcoded RSA-1024 or SHA-1 in the past. The migration to PQC

is an opportunity to build more flexibility: use abstract crypto APIs, negotiate algorithms, support multiple key types, etc. That way, if five years from now you need to swap Kyber for something else (imagine a breakthrough attack, or maybe a new standard that's even better), it's a configuration change or minor update, not a redesign. Some frameworks like TLS already do this well; others, like some custom protocols, might need refactoring.

- **Testing and Validation:** PQC algorithms should be thoroughly tested in your specific environment. For instance, test TLS connections with PQC under high load, test if monitoring tools (IDS/IPS) can handle the larger handshake without false alarms, test backup/restore of PQC keys, etc. Also, test backward compatibility: ensure systems correctly fall back to classical algorithms when a peer doesn't support PQC, and that this doesn't inadvertently drop security when both sides do support PQC (there have been cases in the past where misconfigured negotiation always picked the weaker algorithm).
- **Training and Awareness:** Ensure that developers, security architects, and even procurement teams understand PQC. For example, developers should know that Kyber has different sizes and error considerations (though for users of a library, decapsulation failures are so rare it might never be seen; still, handle any error returns gracefully). Procurement officers should start requiring PQC support in new products (so vendors feel the pressure to include it).
- **Timeline Alignment:** Keep an eye on standards timelines and align your migration. For example, FIPS 203 (Kyber) is final in 2024; you might plan to achieve compliance by 2025 in all new systems. The EU says by 2030 critical infra

should be PQC-protected; an energy company might set a goal to have all control systems upgraded by then. The NSA's roadmap requires NSM 10 compliance and likely says by ~2035 national security systems must use only quantum-resistant algorithms. These timelines provide hard targets to work back from.

- **Dual Key Infrastructure:** During transition, you may need to run two parallel PKIs – one for classical, one for PQC, or an integrated hybrid PKI. Manage this carefully to avoid confusion. One approach is issuing cross-certificates between classical and PQC credentials for the same identity, so that either can be accepted. Another approach is use of composite keys where one certificate actually contains two public keys (but that's still experimental). Whichever, ensure revocation, renewal, etc., processes account for both.
- **Monitoring and Adaptation:** As PQC is deployed, monitor for any anomalies. Are there interoperability issues causing handshakes to fail? Are any devices struggling performance-wise? Also monitor the cryptographic research community – if any new potential weakness is found (e.g., an attack that reduces Kyber's security by some bits), you might consider moving to a higher parameter set (like from Kyber512 to Kyber768) or adding an alternate KEM as a backup.
- **Use Trusted Implementations:** Use well-vetted libraries for PQC (many exist in open source and from established vendors). These libraries should have protections against side-channels and be up-to-date with any improvements. Given that PQC is newer, it's wise not to roll your own. Also consider using implementations that have been tested

in interoperability events (NIST and others hold "PQC hackathons" to test if one library's Kyber ciphertext can be decrypted by another's, etc.).

The migration to post-quantum cryptography is indeed a significant but manageable challenge. It can be compared to the Y2K effort in scope – a lot of systems to touch, but with global awareness and collaboration, it can be done proactively rather than reactively. As one NSA cybersecurity director put it, *"The key is to be on this journey today and not wait until the last minute"*. Organizations that plan and execute their transition will ensure that their data and communications remain secure in the face of the coming quantum era, whereas those that delay may find themselves scrambling later or, worse, see long-term confidential data exposed retrospectively.

11

Kyber Cheat Sheet

What Kyber is

Kyber is a way for two computers to agree on a shared secret key over the internet, even if someone is listening. That shared key is then used for fast symmetric encryption (like AES or ChaCha20). Kyber is designed to stay safe even against future quantum computers.

The big idea

Kyber hides a secret inside math that looks like a simple "equation," but it has a built-in fuzziness (noise). Without the secret key, the noise makes it insanely hard to work backwards and recover the secret.

The "objects" Kyber uses

- Think of a **polynomial** as a list of 256 numbers.
- Those numbers wrap around after a certain size (like a clock). The wrap value is **3329**.
- Kyber does lots of "multiply and add" operations on these

lists.

Why that's hard to break

Kyber's security comes from a family of problems called **lattice problems**. You do not need the deep theory to understand the punchline:

- If there were no noise, an attacker could solve the equations quickly.
- The noise turns it into something like "solving a jigsaw puzzle with pieces that are slightly warped." For big puzzles, it becomes practically impossible.

What Kyber does during a handshake

1. **Key generation (server side)**

- Make a public key (share it with the world).
- Make a secret key (keep it private).

1. **Encapsulation (client side)**

- The client uses the server's public key to create:
- a "locked box" message (ciphertext),
- and a shared secret key (kept by the client).
- The client sends the locked box to the server.

1. **Decapsulation (server side)**

- The server uses its secret key to open the locked box and get the same shared secret key.

Now both sides have the same secret key, without ever sending it directly.

Why Kyber is fast

Multiplying long lists of numbers is expensive if you do it the slow way. Kyber uses a speed trick called the **NTT**, which is basically the "fast multiplication method" for these polynomials (similar spirit to how FFT speeds up signal processing).

What to remember

- Kyber is a quantum-resistant way to agree on a shared secret key.
- It works by mixing secrets with carefully controlled randomness (noise).
- Most of the work is fast list-math plus hashing.

CPU and GPU implementation cheat sheet in plain English (one page)

CPU (normal computer processor): how to make Kyber fast

What costs time

Kyber spends most of its time doing two things:

1. **Hashing / SHAKE** (a special hash used like a random byte generator)
2. **Polynomial math** (multiply/add lists of 256 numbers quickly using NTT)

How to speed it up

- Write the core in a fast systems language: **C, C++, or Rust**.

- Use CPU "vector instructions" (SIMD) so one instruction processes many numbers at once:
- On Intel/AMD: **AVX2** (and sometimes AVX-512)
- On ARM: **NEON** (and sometimes SVE)

Do this in real libraries

- Detect the CPU features at startup and pick the best optimized version automatically.
- Avoid slow things in the hot loop:
- avoid extra memory allocations,
- keep data aligned in memory,
- do fewer "mod" operations (use fast reduction tricks instead).

Security rule

Even when optimizing, do not create timing leaks:

- Do not branch on secret data ("if secret bit then…").
- Do not index arrays using secret values.
- When checking if something matches, compare in constant time.
- This matters because attackers can sometimes measure tiny timing differences.

GPU (graphics processor): when it helps and how to use it

The truth

A GPU can be faster only when you are doing **lots of Kyber operations at once**.

If you are doing one handshake, the CPU is usually faster

because:

- sending work to the GPU has overhead,
- launching GPU kernels has overhead.

When GPU wins

GPU wins when you have a big batch, like:

- a server doing thousands of handshakes per second.

What to put on the GPU

The best GPU job is the big repeated math:

- do many NTTs at once,
- do many "multiply and add" steps at once.

How to organize GPU work

- Give each polynomial to a "block" of GPU threads.
- Use GPU shared memory to hold the 256 numbers so threads can work fast.
- Keep the work fused: avoid copying results back and forth between CPU and GPU too often.

Hybrid strategy that often works best

- CPU does hashing and sampling (because it's already good at it).
- GPU does big batches of polynomial math.
- CPU finishes packing, checks, and final key steps.

12

Conclusion

Encryption is not a single algorithm. It is a chain of decisions. The primitive matters, but the interface matters more than most people admit. The performance profile matters because it shapes what developers will actually deploy. The threat model matters because it determines what "secure" even means for your system.

Kyber and the broader post-quantum transition represent a maturity test for our industry. We are moving from an era where public-key cryptography felt "solved" to one where we must actively manage cryptographic agility, migration planning, and long-term confidentiality. That shift is not merely technical. It is organizational. It requires inventories, governance, disciplined key management, and the ability to update systems safely.

If you take one practical lesson from this book, let it be this: cryptography is a safety-critical subsystem. Treat it with the same seriousness you would treat brakes in a car or control software in a plane. Verify assumptions. Measure performance. Eliminate side channels. Test failure modes. Prefer boring

correctness over clever speed. Then optimize carefully and prove you did not break anything.

Epilogue

A strange thing happens when you truly internalize security engineering: you stop asking whether a system is "secure" in the abstract, and you start asking what it is secure against, for how long, under which conditions, and at what operational cost.

That mindset turns fear into method.

Post-quantum cryptography is not the end of the story. It is a checkpoint. New primitives will arrive, hardware will evolve, attacks will change, and new mistakes will be invented by future versions of ourselves. The correct response is not panic and not complacency. It is disciplined adaptability.

Build systems that can rotate keys without drama. Build protocols that can swap primitives without rewriting everything. Build teams that can read their own dependencies and understand the consequences. That is what it means to engineer for the long run.

Afterword

Security is often treated like a final coat of paint. In reality, it is closer to structural engineering. It is the set of constraints that keeps the system standing when the weather gets violent.

If you are implementing Kyber or any post-quantum primitive, treat your implementation like a product that will be attacked by skilled professionals who have time, money, and patience. Assume they will measure timing. Assume they will target caches. Assume they will exploit error messages. Assume they will read your logs. Assume they will find the one line of code you thought "didn't matter."

Then do the sane thing: write the code as if you will be forced to explain it to a skeptical expert with the exploit in hand.

Because one day, you might be.

Kind regards,
Paul George Savluc